Spring 2008

Parallel MPI/FORTRAN Finite Element Symmetrical/Unsymmetrical Domain Decomposition

Siroj Tungkatara
Old Dominion University

PARALLEL MPI/FORTRAN FINITE ELEMENT

SYMMETRICAL/UNSYMMETRICAL DOMAIN DECOMPOSITION

by

Siroj Tungkahotara
B.Eng., March 1997, Chulalongkorn University
M.E., May 2001, Old Dominion University

A Dissertation submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

CIVIL ENGINEERING

OLD DOMINION UNIVERSITY
May 2008

Approved by:

_____
Duc T. Nguyen (Director)

_____
Zia Razzaq (Member)

_____
Gene Hou (Member)

_____
Chuh Mei (Member)

ABSTRACT

PARALLEL MPI/FORTRAN FINITE ELEMENT

SYMMETRICAL/UNSYMMETRICAL DOMAIN DECOMPOSITION

Siroj Tungkahotara

Old Dominion University, 2008

Director: Dr. Duc T. Nguyen

MPI/FORTRAN finite element analysis software based on Domain Decomposition (DD) formulas has been developed in this work. Efficient input data storage/data communication schemes, domain partitioning, fast symbolical and numerical sparse assembly, symmetrical/unsymmetrical sparse solver and robust symmetrical/unsymmetrical iterative solvers algorithms are all incorporated into the developed code. Parallel Precondition Conjugated Gradient (PCG) and Flexible Generalized Minimum Residual (FGMRES) are developed. Efficient computational techniques used in the developed code are explained. Numerical performance and the accuracy of the developed code are conducted on acoustic examples with medium to large grid sizes. The results obtained from ODU Wilbur cluster (under parallel processing computer environments) have revealed the super-linear speedup in 3-D symmetrical acoustic examples. The robustness and the minimum in-core memory usage of the code are also observed.

This dissertation is dedicated to my family

for their support, love, understanding and encouragement.

ACKNOWLEDGMENTS

I would like to express my sincere gratefulness to my advisor, Prof. Duc T. Nguyen for his support and advice during the course of this research. Several subroutines used in this work have been based on his textbooks and from his earlier research software. I would also like to thank other members of the dissertation committee, Prof. Zia Razzaq, Prof. Gene Hou and Prof. Chuh Mei for their helpful suggestions and comments during the early stages of my study.

I acknowledge the financial support and large-scale acoustic codes provided by Dr. Willie R. Watson at NASA Langley Research Center. My appreciation also goes to ODU/OCCS for the excellent consulting services, troubleshooting and computer facilities. Also, I would like to thank Dr. Hakizumwami Birali Runesha, Dr. J. Qin and Prof. Subramaniam D. Rajan for their advice, clarification and recommendations on portions of my research work.

Lastly, I truly appreciate my family for their support, understanding and encouragement.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION[1]

1.1 Overview

In the past decades, finite element method (FEM) has been playing a major role in many engineering disciplines, such as structural analysis, fluid dynamic, heat transfer, structural dynamic, structural optimization, groundwater flow, etc. Both linear and nonlinear analysis, both statics and time dependent problems can all be treated under a general, unified FEM. In sequential finite element step-by-step procedures, the equation solver phase (to solve a large system of simultaneous linear equations) consumes most computational resources (in terms of computational time and memory). Furthermore, using sequential equation algorithms and their associated solver software will limit the ability to solve large-scale problems on distributed memory computers, which are widely available in existing modern computer hardware markets.

For the above mentioned reasons, the equation solver topic has been of interest to numerous researchers (Amestoy, Duff and L'Excellent, Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers; Amestoy, Duff and L'Excellent, Mumps Multifrontal Massively Parallel Solver Version 2.0.; Farhat and Roux, Implicit Parallel Processing in Structural Mechanics; Saad, A Flexible Inner-Outer Preconditioned GMRES Algorithm), and the goals for these researches are to develop efficient numerical algorithms for minimizing time and resources used by the solvers while maintaining high accuracy of the solutions. To achieve these goals, vector and/or parallel sparse

---

[1] The journal model used is Modern Language Association, MLA.

computational techniques (Razzaq, Prasad and Darbhamulla) have all been utilized for the application coder. Vector computational techniques include the efficient usage of cache memory, storing data in sparse format, reordering the coefficient "stiffness" matrix to reduce fill-in terms, unrolling strategies, etc. On the other hand, parallel computational techniques need to be designed for minimizing communication time amongst processors, balancing the workload on each processor, and redesigning sequential algorithms to optimize its performance in parallel computer environments.

Nowadays, the availability of multi-core processors for home and small business users combined with fast network switching can substantially reduce the cost of building a High Performance Computing (HPC) cluster. Basically, this is a distributed memory machine in which each processor has its own amount of memory. Although some cluster setups will allow each processor to access memory of other processors, the transfer rate of data through the network switching will drastically decrease; due to the fact that the bandwidth of transferring data over the network is much lower than inter-communication within the processor itself. In summary, large-scale problems' solutions can tremendously benefit from using today's processor architecture technologies and carefully designed parallel equation solvers.

1.2 Review of previous work

Domain decomposition (DD) formulation has attracted many researchers since it was introduced in 1963 (Przemieniecki). The method was originally used to divide the structure into substructures, and each substructure was computed separately in a "sequential" fashion. This is done since the computational resources of "sequential" computers at that time were inadequate to handle the computation of the entire structure

at once. Moreover, each substructure can have different types of analysis. For this method, all subdomains are still connected to each other. Nodes in the domain are distinguished as boundary nodes, which connect two or more subdomains together, and interior nodes, which belong to one subdomain. The boundary nodal displacements are then obtained by solving the Schur complement equation. After that, interior displacements can be obtained for each subdomain. This method is also known as primal domain decomposition formulation. In Finite Element Tearing & Interconnecting, FETI (Farhat and Roux, Implicit parallel processing in structural mechanics), there is no distinction between boundary and interior nodes, and all the subdomains are completely disconnected initially. Then, the interfaced displacements among subdomains will be incorporated later as the dual unknown Lagrange multipliers. In FETI-DP (Farhat, Lesoinne and LeTallec, "FETI-DP: a Dual-Primal Unified FETI Method - Part I: A Faster Alternative to the Two-Level FETI Method"), only enough nodes or degrees of freedom that remove rigid body translations/rotations are required to connect 2 adjacent subdomains. Such nodes are called the "corner" nodes, and the other nodes are called the "remaining" (boundary and interior) nodes.

The Domain Decomposition method has now been widely accepted among research communities due to the fact that it has high level of scalability, and it can be effectively implemented on modern computer architectures. In addition, high performance computing clusters available today are mainly distributed memory machine clusters (TOP500.Org). Thus, developing algorithms for the clusters has become a challenging task since the available memory on each processor and the communication bandwidth between processors are limited on such machines.

In 1992, MPI1 standardization (MPI: A Message-Passing Interface Standard) was proposed in an effort to improve the efficiency of inter-communication in HPC clusters. Basically, MPI is an application programming interface that allows C, C++, FORTRAN 77, FORTRAN 90, etc. bindings. The goals of MPI are to create reliable communication interface, allow efficient communication in both heterogeneous and non-heterogeneous environments, and allow multi-platform implementations. In 1995, MPI standard 2.0 (MPI2: Extensions to the Message-Passing Interface) was announced in order to extend the capability of MPI1 such as dynamic processes, one-sided communication, and parallel I/O.

Balancing the workload on each processor is one of the research areas that can be improved. Without balancing of the domain (or mesh) partitioning, some processors will have more workload than the others. ParMETIS (Karypis, Schloegel and Kumar) is an automatic mesh partitioning algorithm that partitions unstructured mesh in such a way to minimize the number of edges cut by the partitioning. In other words, ParMETIS attempts to reduce the communications between subdomains during the computational phases. Not only can ParMETIS minimize the communications, but also partitioning the mesh such that the partitioned subdomains are well balanced. Another capability of ParMETIS is that it can be used to reorder the coefficient matrix for minimizing fill-ins terms during the numerical factorization phase. Furthermore, since ParMETIS stores adjacency information of the structure in distributed compressed storage scheme (in row-wise format) among processors, large-scale problems can be efficiently partitioned on distributed memory machines.

Sparse storage scheme has been widely used in order to avoid the unnecessary

computational of zero terms during the equation solver phase. Using this storage scheme, the coefficient matrix is represented by 1-dimensional arrays, and only non-zero locations and values are stored. On the other hand, traditional bandwidth and skyline storage schemes still store "some" zero values of the matrix for possible usage during the numerical/factorized computation phase. As a result, sparse storage scheme has substantial advantages over bandwidth and skyline storage schemes since zero terms are not stored, nor computed in the numerical process (Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications).

As noted by Sadd in 2000 (Saad, "Yousef Saad – Books"), direct sparse solution methods were more preferable (especially in the earlier years) to iterative methods due to their robustness and predictable behavior. On the other hand, iterative methods were special-purpose in nature and were developed for certain applications where their efficiency relied on many problem-dependent parameters. Recently, the combination of good preconditioning and Krylov subspace iterations have provided better behavior and efficiency to iterative methods. As a result, the popularity of the efficient iterative methods has gradually grown owing to their capability to solve large-scale systems and the comparability of their quality to the direct sparse solution methods.

1.3 Objectives and scope

A portable implementation based on primal domain decomposition formulation used in this work is discussed. Domain decomposition (DD) formulation is utilized to solve large-scale engineering/science applications, for both "symmetrical" and "unsymmetrical" systems. While several researchers have reported and documented their successful results for solving large-scale "symmetrical" equations, much less successful

stories for "unsymmetrical" cases have been reported in existing literatures. Additional goals of this work are to develop a code based on DD formulation written in generic FORTRAN 90, which can run on any platform (shared or distributed memory computer environment), and has the following features/capabilities:

Developing robust FEA software using domain decomposition formulas for both symmetrical and unsymmetrical systems.

Developing a distributed storage scheme for solving large-scale problems on a distributed memory machine.

Developing a robust/automatic domain partitioning scheme to break the entire domain's information into subdomains' information.

Utilizing ParMETIS library for automatic mesh partitioning on large-scale problems.

Developing efficient communication schemes using MPI library for both shared and distributed memory computer systems.

Utilizing reordering scheme (METIS) for reducing fill-ins during factorization phase.

Developing parallel assembly of the entire domain on the basis of domain decomposition context.

Developing robust iterative solvers to solve boundary degrees of freedom for both symmetrical and unsymmetrical systems.

Developing user friendly interfaces for adding new finite element into the current code.

In chapter 2, the primal DD formulation is briefly summarized. The efficient

computer implementation procedures, including mixed direct-iterative solvers and efficient parallel procedures for matrix-vector operations in DD formulation are discussed. Moreover, imposing multi-point constraints (MPC) in DD formulation is also discussed in this chapter.

In Chapter 3, sparse matrix computations are reviewed. These will include sparse data storage schemes, sparse reordering for minimizing fill-in terms, sparse assembly, sparse equation solver, and unrolling techniques.

In chapter 4, Preconditioned Conjugate Gradient (PCG) and Flexible Generalized Minimum Residual (FGMRES) used to solve the system of symmetrical and unsymmetrical equation, respectively, are discussed. The pseudo serial codes of both algorithms are provided in this chapter. Then, the data storage scheme of PCG is explained, and the pseudo parallel codes of both algorithms are summarized. Moreover, three preconditioning techniques utilized for domain decomposition context are also covered in this chapter.

In chapter 5, numerical performance (in terms of the solution accuracy, computer memory requirements and efficiency) of the developed code for several large-scale numerical examples has been conducted. Both symmetrical and unsymmetrical solvers are used on the first example, which is a 3-D acoustic finite element model without flow (symmetrical system). Then, the unsymmetrical solver is used on the second example, which is a 2-D acoustic finite element model with flow (unsymmetrical system).

In chapter 6, implementation techniques used in the code are discussed. These will include data partitioning of user's input for large-scale problems, data preparing for ParMETIS subroutine, post processing of ParMETIS's result to obtain subdomains'

information, obtaining non-zero locations in K_{bi} and K_{ib} matrices, and numerical assembly for K_{bb}, K_{bi} and K_{ii}.

Finally, conclusions and suggestions for future research works are mentioned in chapter 7.

1.4 Assumptions

The software developed in this work is used to analyse the systems of symmetrical and unsymmetrical linear static, and it has the capabilities to handle both real and complex arithmetic. In addition, two acoustic finite element types have been incorporated into the code, and additional finite element types can be integrated into the code. In contrast, only 1 element type and 1 material set in the problem have been fully tested in the developed software, and the maximum number of nodes per element is set to be 8.

CHAPTER II

FINITE ELEMENT DOMAIN DECOMPOSITION (DD) FORMULATION

2.1 Introduction

From the static finite element equilibrium equation,

$$[K]\cdot\vec{x} = \vec{f} \tag{2.1}$$

where,

$[K]$ = Stiffness matrix

$\vec{x}$ = displacement vector

$\vec{f}$ = load vector

Equation (2.1) can be partitioned as:

$$\begin{bmatrix} K_{BB} & K_{BI} \\ K_{IB} & K_{II} \end{bmatrix} \cdot \begin{Bmatrix} x_B \\ x_I \end{Bmatrix} = \begin{Bmatrix} f_B \\ f_I \end{Bmatrix} \tag{2.2}$$

Where subscripts B and I denote Boundary and Interior parts of the stiffness matrix, respectively.

The first and the second parts of equation (2.2) could be expressed, respectively, as:

$$K_{BB}\cdot x_B + K_{BI}\cdot x_I = f_B \tag{2.3}$$

, and

$$K_{IB}\cdot x_B + K_{II}\cdot x_I = f_I \tag{2.4}$$

From equation (2.4), one obtains:

$$\vec{x}_I = [K_{II}]^{-1} \cdot \left(\vec{f}_I - K_{IB} \cdot \vec{x}_B\right) \tag{2.5}$$

Substituting (2.5) into (2.3), one gets:

$$K_{BB} \cdot x_B + K_{BI} \cdot [K_{II}]^{-1} \cdot \left(\vec{f}_I - K_{IB} \cdot \vec{x}_B\right) = f_B \tag{2.6}$$

which can be written as:

$$\left[K_{BB} - K_{BI} \cdot K_{II}^{-1} \cdot K_{IB}\right] \cdot \vec{x}_B = \left(f_B - K_{BI} \cdot K_{II}^{-1} \cdot f_i\right) \tag{2.7}$$

or

$$\overline{K}_B \cdot \vec{x}_B = \overline{F}_B \tag{2.8}$$

Where $\overline{K}_B$ is the effective boundary stiffness which can be expressed as:

$$\overline{K}_B = K_{BB} - K_{BI} \cdot K_{II}^{-1} \cdot K_{IB} \tag{2.9}$$

and $\overline{F}_B$ is the effective boundary load, which can be expressed as:

$$\overline{F}_B = f_B - K_{BI} \cdot K_{II}^{-1} \cdot f_i \tag{2.10}$$

$\overline{K}_B$ is also called Schur complement matrix.

Once the boundary displacement vector $\vec{x}_B$ is obtained from equation (2.8), the interior displacement vector $\overline{x}_I$ could be solved by equation (2.5).

For very large-scale problems, the original stiffness matrix could be partitioned into several smaller sub-domains. For r^{th} sub-domain, equations (2.5), (2.9) and (2.10) could

be expressed, respectively, as:

$$\vec{x}_I^{(r)} = \left[K_{II}^{(r)}\right]^{-1} \cdot \left(\vec{f}_I^{(r)} - K_{IB}^{(r)} \cdot \vec{x}_B^{(r)}\right) \tag{2.11}$$

$$\overline{K}_B^{(r)} = K_{BB}^{(r)} - K_{BI}^{(r)} \cdot \left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)} \tag{2.12}$$

$$\overline{F}_B^{(r)} = f_B^{(r)} - K_{BI}^{(r)} \cdot \left[K_{II}^{(r)}\right]^{-1} \cdot f_i^{(r)} \tag{2.13}$$

Then, the system effective boundary stiffness matrix ($\overline{K}_B$) and the system effective boundary load can be assembled as:

$$\overline{K}_B = \sum_{r=1}^{nsub} \overline{K}_B^{(r)} \tag{2.14}$$

$$\overline{F}_B = \sum_{r=1}^{nsub} \overline{F}_B^{(r)} \tag{2.15}$$

where nsub is the number of total subdomains

Then, the system boundary displacements can be obtained by using equation (2.8). Finally, the subdomains' interior displacements can be solved by using equation (2.11).

2.2 Using mixed direct-iterative solvers in DD formulation

For very large-scale problems when the numbers of subdomains' boundary degrees of freedom are large, the triple-product operations involved in equation (2.12) are quite expensive because:

1. $\left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)}$ requires performing backward substitution equal to the number of subdomain's boundary degrees of freedom which is normally large. Although $K_{ii}^{(r)}$ and

$K_{IB}^{(r)}$ are sparse matrices, the result of $\left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)}$ operations will be a dense or nearly dense matrix.

2. The result from the previous step, which is a dense matrix, will require sparse matrix times dense matrix operations to perform $K_{BI}^{(r)} \cdot \left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)}$.

In mixed direct-iterative solvers, the triple product in equation (2.12) will never be formed explicitly. Therefore, mixed direct-iterative solvers are recommended to obtain the system boundary degrees of freedom in equation (2.8).

2.3 Efficient parallel procedures for matrix times vector in DD formulation

To solve the boundary degrees of freedom in equation (2.8) using a mixed-iterative solver, there are several places involved with a matrix times a known vector. In this case, $\overline{K}_B$ is the matrix that needs to multiply with a known vector, $\vec{v}$. Since $\overline{K}_B$ is the summation of $\left(\overline{K}_B\right)^{(r)}$ of each subdomain, the result of $\overline{K}_B \cdot \vec{v}$ could be obtained simultaneously on the processes.

Using,

$$\overline{K}_B = \sum_{r=1}^{nsub} \overline{K}_B{}^{(r)} \tag{2.14}$$

and substituting equation (2.12) into equation (2.14), one obtains

$$\overline{K}_B = \sum_{r=1}^{nsub} \left[K_{BB}^{(r)} - K_{BI}^{(r)} \cdot \left[K_{II}^{(r)} \right]^{-1} \cdot K_{IB}^{(r)} \right] \tag{2.16}$$

Post-multiplying equation (2.16) with a known vector $\vec{v}$, one obtains

$$\bar{K}_B \cdot \vec{v} = \sum_{r=1}^{nsub} \left[K_{BB}^{(r)} - K_{BI}^{(r)} \cdot \left[K_{II}^{(r)} \right]^{-1} \cdot K_{IB}^{(r)} \right] \cdot \vec{v} \tag{2.17}$$

which could be expressed as:

$$\bar{K}_B \cdot \vec{v} = \sum_{r=1}^{nsub} \left[\left[K_{BB}^{(r)} - K_{BI}^{(r)} \cdot \left[K_{II}^{(r)} \right]^{-1} \cdot K_{IB}^{(r)} \right] \cdot \vec{v}^{(r)} \right] \tag{2.18}$$

where $\vec{v}^{(r)}$ is the reduced size of the known vector $\vec{v}$ corresponding to the subdomain's boundary degrees of freedom.

Since all matrices and vector in the right hand side of equation (2.18) are in subdomain level, the operations could be performed independently on each processor. Then, all processors will combine their results at the end to find the resulting vector. Furthermore, the operations involved in the right-hand-side of equation (2.18) should be preceded from right to left (for computational efficiency purposes). To obtain the product of equation (2.18), one should follow the steps below.

1. Each processor computes $K_{IB}^{(r)} \cdot \overrightarrow{v^{(r)}}$ and stores in T_1 array of size nidof.
2. Each processor computes $\left[K_{II}^{(r)}\right]^{-1} \cdot \vec{T_1}$ and stores in T_2 array of size nidof.
3. Each processor computes $K_{BI}^{(r)} \cdot \vec{T_2}$ and stores in T_1 array of size nbdof.
4. Each processor computes $K_{BB}^{(r)} \cdot \overrightarrow{v^{(r)}}$ and stores in T_2 array of size nbdof.
5. Each processor computes $T_2 - T_1$ and stores in T_2 array of size nbdof.
6. Since T_2 is a local product, each processor creates a global vector of T_2.
7. All processors collect and combine the products from all processors to get the final result of equation (2.18).

The procedure to compute equation (2.15) is similar to the procedures to compute equation (2.18), so the procedures discussed here can be used to get the result of equation (2.15).

2.4 Detailed step-by-step procedures for mixed direct-iterative with DD formulation

2.4.1. User input

Before calling the Domain Decomposition Finite Element Analysis subroutine, the user will provide the problem information such as, global element connectivity, nodes coordinates, external load information, material properties, multi-point constraints information, Dirichlet boundary conditions and etc. The input format will be clearly explained in appendix A.1.

Please note that all the information provided by the user will be the global level information. Then, this information will be broken up into several parts during the call to ddfea subroutine.

However, for large-scale 3-D acoustic problems (i.e. more than 25 million degrees of freedom), the entire system input information might require more than half of the memory available for each processor on a distributed memory computer system. Therefore, the input information has to be partitioned and stored among the processors by the computer code before calling DDFEA subroutine, and this step is discussed in chapter 6.1.

2.4.2. Domain breaking phase

To break the entire domain, ParMETIS (Karypis, Schloegel and Kumar) is used to partition the domain into several parts depending on how many processors are

being used. In this version of the computer code, each subdomain is handled by one processor. The result from ParMETIS will only provide the information about which owner (or subdomain) each degree of freedom belongs to. For example, the ParMETIS result will look like the example shown below.

$$MET = [1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,3]$$

It can be seen that, for this particular example, degrees of freedom 1 to 6, 7 to 12 and 13 to 18 originally belonged to subdomain 1, 2 and 3, respectively. Please note that ParMETIS cannot identify the global (system) boundary degrees of freedom. Therefore, post processing of ParMETIS result is required to find the subdomains' information, such as boundary degrees of freedom list, interior degrees of freedom list, elements' information, nodes' coordinates, material properties, multi-point constraints information, Dirichlet boundary conditions, external load conditions, etc.

The following 3 key steps are involved in this phase, which will be clearly explained in chapter 6.2 and 6.3:

ParMETIS data preparing to save memory

Obtaining boundary degrees of freedom from ParMETIS result

Incorporating multi-point constraints equations in DD formulations.

It should be also noted that the output information from this phase will be reduced from the global system size to subdomains' size.

2.4.3. Subdomains' connectivity phase

Utilizing the element information output from the previous step, three groups of

element connectivity of each subdomain are obtained in this step since three submatrices, which are K_{BB} , K_{BI} , K_{IB} and K_{II} , are constructed on each subdomain later in the next few steps. The first one is the element connectivity of the elements associated with only local boundary nodes. The second one is the element connectivity of the elements associated with only interior nodes. The last one is the element connectivity of the elements associated with both boundary and interior nodes. Only the first two element connectivities will be the output of this phase. The two CSR format arrays, *IABI* and *JABI* , representing non-zero locations of K_{bi} and K_{ib} will also be the output from this phase. The detailed steps of how to obtain non-zero locations of K_{bi} and K_{ib} will be clearly explained in chapter 6.4.

2.4.4. Subdomains' reordering phase

After obtaining element connectivity of the elements associated with interior degrees of freedom from the previous step, the adjacency arrays of interior elements are created as the input for reordering algorithms (Metis or Nested Disection algorithms can be selected). As discussed in Chapter 3.2, this step is performed in order to reduce the number of fill-in terms of the factorized matrix.

2.4.5. Subdomains' assembly phase

In this step, the matrices' information is obtained for both non-zero locations and their numerical values. Matrix K_{bb} is represented by arrays *IABB* , *JABB* , *ADBB* and *ANBB* . Matrix K_{bi} is represented by arrays *IABI* , *JABI* and *ANBI* . Also, matrix K_{ii} is represented by array *IAII* , *JAII* , *ADII* and *ANII* . For the unsymmetrical problem, there are 3 additional arrays, which are *ANBB2* , *ANIB* and

ANII2 required to represented K_{bb}, K_{bi} and K_{ii}, respectively. In addition, the sizes of all matrices are in subdomain level. The sizes of these arrays are shown in Table 2.1, where nbdof is the number of subdomain's boundary degrees of freedom, and nidof is the number of subdomain's interior degrees of freedom.

2.4.6. Subdomains' factorization phase

Before performing equation (2.18) in the iterative solver phase, K_{ii} matrix needs to be firstly factorized. Using the sparse factorization strategies discussed in chapter 3.5, 3.8 and 3.9, the factorized matrix can be obtained and represented in the arrays *IU*, *JU*, *DI* and *UN* for symmetrical matrices, and *UN2* as an additional matrix for unsymmetrical matrices. The sizes of these arrays are represented in Table 2.2.

Table 2.1: Size of the subdomain's matrices

Array	Size
IABB	nbdof+1
JABB, ANBB and ANBB2	IABB(nbdof+1)-1 = ncoef1bb
ADBB	nbdof
IABI	nbdof+1
JABI, ANBI and ANIB	IABI(nbdof+1)-1 = ncoef1bi
IAII	nidof+1
JAII, ANII and ANII2	IAII(nidof+1)-1 = ncoef1ii
ADII	nidof

Table 2.2: Size of subdomain factorized matrix

Array	Size
IU	nidof+1
JU, UN, and UN2	IU(nidof+1)-1 = ncoef2ii
DI	nidof

2.4.7. Subdomains' boundary displacements solution phase

As discussed in chapter 2.2, a mix-iterative solver is recommended to solve for the displacements of boundary dofs in equation (2.8). Also, since the mixed-iterative solvers involve with K_B matrix times a known vector, the efficient matrix times vector subroutine should also be considered in this phase. This step has already been explained in chapter 2.3. In this work, Pre-conditioned Conjugate Gradient is selected to solve the system of symmetrical matrices, and FGMRES(m) is selected to solve the system of unsymmetrical matrices. These two iterative solvers will be discussed in chapter 4. Upon successful completion of this phase, the boundary degrees of freedom displacements, $\vec{X}_B$, of size nbdofall, total number boundary degrees of freedom, are obtained.

2.4.8. Subdomains' interior displacements solution phase

Utilizing the boundary degrees of freedom displacements obtained in the previous step, each processor performs equation (2.11) in order to acquire the subdomain's interior degrees of freedom. Upon successful completion of this phase, the interior degrees of freedom displacements, $\vec{X}_I$ (of size nidof = number of subdomain's interior degrees of freedom) are obtained.

2.4.9. Error checking phase

From equation (2.2), one has:

$$\begin{bmatrix} K_{BB} & K_{BI} \\ K_{IB} & K_{II} \end{bmatrix} \cdot \begin{Bmatrix} x_B \\ X_I \end{Bmatrix} = \begin{Bmatrix} f_B \\ f_I \end{Bmatrix} \qquad \text{(2.2), repeated}$$

which can be expressed as

$$K_{BB} \cdot x_B + K_{BI} \cdot x_I = f_B \qquad \text{(2.3), repeated}$$

and

$$K_{IB} \cdot x_B + K_{II} \cdot x_I = f_I \qquad \text{(2.4), repeated}$$

After acquiring $\vec{X}_B$ and $\vec{X}_I$ from the previous discuss sections, the residual of equation (2.2) can be written as:

$$\begin{bmatrix} f_B - K_{BB} \cdot x_B + K_{BI} \cdot x_I \\ f_I - K_{IB} \cdot x_B + K_{II} \cdot x_I \end{bmatrix} = \begin{bmatrix} r_B \\ r_I \end{bmatrix} = [r] \qquad (2.19)$$

where r_B and r_I are residual vectors of boundary and interior parts, respectively.

From equation (2.19), r_I in the second equation could be independently done by each processor since there is no coupling of K_{IB} and K_{II} between subdomains. For the first equation, however, each processor calculates its own boundary residual, r_B, and the results need to be combined with other processors.

The absolute error norm is defined as

$$\|r\| = \sqrt{\vec{r} \cdot \vec{r}}$$

And the relative error norm is defined as

$$\frac{\|r\|}{\|f\|} = \frac{\sqrt{\vec{r} \cdot \vec{r}}}{\sqrt{\vec{f} \cdot \vec{f}}}$$

2.4.10. Inversion of the displacements from subdomains to the original numbering system

Before exiting the program, all displacements obtained by the processors are collected by the master procesor (processor 0 in this version of the code). The mapping array between the global system and subdomains' system is used along with the Metis reordering information.

2.5 Multi point constraints in DD formulation

To demonstrate the multi-point constraints capability in the domain decomposition formulation, a 4-node, 5-element example with "inclined" roller support (at joint 2) is introduced in Figure 2.1.

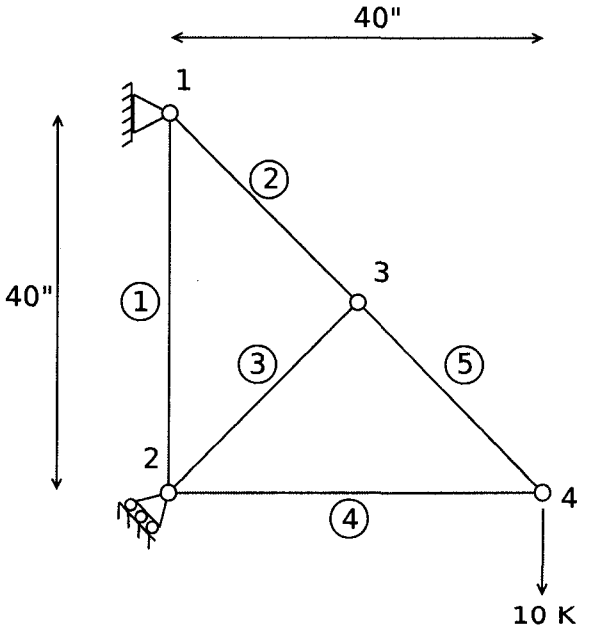


Figure 2.1: 4-node, 5-element truss example with an inclined support

The MPC equation at support 2 could be expressed as:

$$c_3 x_3 + c_4 x_4 = D \tag{2.20}$$

where x_3 is the horizontal displacement, and x_4 is the vertical displacement at node 2. Also, c_3 , c_4 and D are known constants.

The MPC equation (2.20) can be generalized to the following form:

$$\begin{aligned}
&c_{1,1}x_1 + c_{1,2}x_2 + \ldots + c_{1,i}x_i + c_{1,j}x_j + \ldots + c_{1,n}x_n = D_1 \\
&c_{2,1}x_1 + c_{2,2}x_2 + \ldots + c_{2,i}x_i + c_{2,j}x_j + \ldots + c_{2,n}x_n = D_2 \\
&\qquad\qquad\qquad\qquad \vdots \\
&c_{i,1}x_1 + c_{i,2}x_2 + \ldots + c_{i,i}x_i + c_{i,j}x_j + \ldots + c_{i,n}x_n = D_i \\
&c_{j,1}x_1 + c_{j,2}x_2 + \ldots + c_{j,i}x_i + c_{j,j}x_j + \ldots + c_{j,n}x_n = D_j \\
&\qquad\qquad\qquad\qquad \vdots \\
&c_{n,1}x_1 + c_{n,2}x_2 + \ldots + c_{n,i}x_i + c_{n,j}x_j + \ldots + c_{n,n}x_n = D_n
\end{aligned} \tag{2.21}$$

where $c_{i,j}$ and D_i are known constants.

From equation (2.1), one recalls:

$$[K]\cdot\vec{x} = \vec{f} \tag{2.1), repeated}$$

The total potential energy of the system of equation (2.1) with MPC in equation (2.20) could be expressed as:

$$Minimizing\ \prod(x) = \frac{1}{2}x^T Kx - x^T f + \frac{1}{2}P(c_3 x_3 + c_4 x_4 - D)^2 \tag{2.22}$$

where P is a big number which, according to (Rajan), is $10^4 \cdot max\left|K_{p,q}\right|$.

The terms appearing inside the right-hand-side parenthesis in equation (2.22) need be

squared to guarantee a positive value (for a proper penalty term). The factor $\frac{1}{2}$ could have been absorbed by the positive, large constant P. However, this factor $\frac{1}{2}$ will be conveniently disappeared when the partial derivative of Π is computed.

From equation (2.22), it is seen that the total potential energy is minimum when $\frac{\partial \Pi}{\partial D} = 0$. The derivative yields the usual total stiffness matrix and right-hand-side vector except the rows and columns associated with x_3 and x_4, in this case. The modified terms of rows and columns 3 and 4 are:

$$\begin{bmatrix} \ddots & \vdots & \vdots & \\ \cdots & k_{3,3} + Pc_3^2 & k_{3,4} + Pc_3c_4 & \cdots \\ \cdots & k_{4,3} + Pc_3c_4 & k_{4,4} + Pc_4^2 & \cdots \\ & \vdots & \vdots & \ddots \end{bmatrix} \tag{2.23}$$

and

$$\begin{bmatrix} \vdots \\ F_3 + PDc_3 \\ F_4 + PDc_4 \\ \vdots \end{bmatrix} \tag{2.24}$$

It is worth taking a closer look at the first row of equation (2.21). In general, the additional terms from the first row of equation (2.21) could be expressed as:

$$\begin{bmatrix} Pc_{1,1}^2 & Pc_{1,1}c_{1,2} & \cdots & Pc_{1,1}c_{1,i} & Pc_{1,1}c_{1,j} & \cdots & Pc_{1,1}c_{1,n} \\ Pc_{1,1}c_{1,2} & Pc_{1,2}^2 & \cdots & Pc_{1,2}c_{1,i} & Pc_{1,2}c_{1,j} & \cdots & Pc_{1,2}c_{1,n} \\ & & & \vdots & & & \\ Pc_{1,1}c_{1,i} & Pc_{1,2}c_{1,i} & \cdots & Pc_{1,i}^2 & Pc_{1,i}c_{1,j} & \cdots & Pc_{1,i}c_{1,n} \\ Pc_{1,1}c_{1,j} & Pc_{1,2}c_{1,j} & \cdots & Pc_{1,i}c_{1,j} & Pc_{1,j}^2 & \cdots & Pc_{1,j}c_{1,n} \\ & & & \vdots & & & \\ Pc_{1,1}c_{1,n} & Pc_{1,2}c_{1,n} & \cdots & Pc_{1,i}c_{1,n} & Pc_{1,j}c_{1,n} & \cdots & Pc_{1,n}^2 \end{bmatrix} \tag{2.25}$$

and

$$\begin{bmatrix} PD_1c_{1,1} \\ PD_1c_{1,2} \\ \vdots \\ PD_1c_{1,i} \\ PD_1c_{1,j} \\ \vdots \\ PD_1c_{1,n} \end{bmatrix} \tag{2.26}$$

Notes: If N represents the size of matrix [K], and n represents the number of MPC equations, n is less than N.

As one can see, the additional terms could be considered a "fictitious, or artificial" element stiffness matrix of the MPC element. In other words, the MPC equations are considered in this work as extra, artificial elements. For each extra, artificial element, the number of nodes is the number of degrees of freedom associated with the extra element, i.e. the MPC equation. Also, the number of degrees of freedom per node of the extra element is 1. Since all MPC equations are treated as elements, they have to be included in the phase to find boundary degrees of freedom in order to avoid the coupling of interior

degrees of freedom between two subdomains.

As a quick example, suppose the following 2 MPC equations need to be implemented:

$$\begin{aligned} 2 \cdot x_3 - 8 \cdot x_{17} + 4 \cdot x_{25} &= -6 \\ -4 \cdot x_8 + 12 \cdot x_{23} &= 5 \end{aligned}$$

Thus, the following "extra, artificial" MPC finite elements need be created:

$$\left[k_{MPC}^{(e=1)}\right] = \begin{bmatrix} P(2)^2 & P(2)(-8) & P(2)(4) \\ P(-8)(2) & P(-8)^2 & P(-8)(4) \\ P(4)(2) & P(4)(-8) & P(4)^2 \end{bmatrix}$$

and

$$\left\{f_{MPC}^{(e=1)}\right\} = \begin{Bmatrix} P(-6)(2) \\ P(-6)(-8) \\ P(-6)(4) \end{Bmatrix}$$

which associate with degree of freedom 3, 17 and 25, respectively.

$$\left[k_{MPC}^{(e=2)}\right] = \begin{bmatrix} P(-4)^2 & P(-4)(12) \\ P(12)(-4) & P(12)^2 \end{bmatrix}$$

and

$$\left\{f_{MPC}^{(e=2)}\right\} = \begin{Bmatrix} P(5)(-4) \\ P(5)(12) \end{Bmatrix}$$

which associate with degree of freedom 8 and 23, respectively.

In this work, the information of MPC equations are stored in nmpcg, IAMPCG, JAMPCG, CMPCG, RMPCG. The description of each variable is explained below.

nmpcg – A number of MPC equations

IAMPCG(nmpcg+1) – An integer array containing the number of degrees of freedom associated with each MPC equation. IAMPCG(i+1)-IAMPCG(i) indicates the number of degrees of freedom associated with i^{th} MPC equation.

JAMPCG(IAMPCG(nmpcg+1)-1) – An integer array containing the list of degrees of freedom associated to MPC equations

CMPCG(IAMPCG(nmpcg+1)-1) – A double complex array containing the list of coefficient values of the degrees of freedom associated in MPC equations

RMPCG(nmpcg) – A double complex array containing the values of right-hand-side of MPC equations.

For the example, the information of MPC equations can be expressed as below.

$$nmpcg = 2$$

$$IAMPCG = [1,4,6]^T$$

$$JAMPCG = [3,17,25,8,23]^T$$

$$CMPCG = [2,-8,4,-4,12]^T$$

$$RMPCG = [-6,5]^T$$

CHAPTER III

SPARSE MATRIX COMPUTATION

3.1 Sparse matrix data formats

The compressed storage in row, CSR format scheme is used in this work. For symmetrical matrices, only diagonal and non-zero terms in the upper triangular part of the matrix will be stored. For any matrix stored using this scheme, the matrix information can be stored in 2 integer arrays (IA and JA) and 2 double real or complex arrays (AD and AN). For unsymmetrical matrices, one more array, which is AN2, will be needed in order to store the lower triangular part of the matrix. The integer array IA of size (N+1), where N is the rank of the matrix, describes the starting index of the first non-zero element of each row in the matrix. The integer array JA(NCOEF), where NCOEF is the total number of non-zero terms in the upper triangular part of the matrix, describes the column numbers associated with non-zero terms of each row of the upper triangular portion of the matrix. The array IA and JA can be demonstrated by the following example.

For a matrix,

$$K = \begin{bmatrix} 2 & 0 & 4 & 0 & 6 \\ 0 & 4 & 0 & 0 & 7 \\ 4 & 0 & 6 & 7 & 0 \\ 0 & 0 & -7 & 8 & 9 \\ -6 & 2 & 0 & -9 & 10 \end{bmatrix}$$

one gets

$$IA = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 5 \\ 6 \\ 6 \end{bmatrix}$$

, and

$$JA = \begin{bmatrix} 3 \\ 5 \\ 5 \\ 4 \\ 5 \end{bmatrix}$$

From this example, it can be seen that there is 1 non-zero term (excluding the diagonal term) in the upper portion of row 2 ($IA(3) - IA(2)$), and the column index of row 2 will be from $JA(3)$ to $JA(3)$ (i.e. from $IA(2)$ to $IA(3)-1$), which is column 5 in this example.

The diagonal and off-diagonal values of K can be described by the double real or complex arrays AD and AN, respectively. From the same example, one gets AD and AN as follows.

$$AD = \begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{bmatrix}$$

$$AN = \begin{bmatrix} 4 \\ 6 \\ 7 \\ 7 \\ 9 \end{bmatrix}$$

Because the example shown here is an unsymmetrical matrix, an extra array AN2 is needed to store the information of the lower triangular part of the matrix. Although the off-diagonal terms of the upper triangular and the lower triangular parts of the matrix K have different values, the column-wise non-zero pattern of the lower triangular part will be identical to the row-wise non-zero pattern of the upper triangular part. In this particular example, AN2 can be obtained as follows.

$$AN2 = \begin{bmatrix} 4 \\ -6 \\ 2 \\ -7 \\ -9 \end{bmatrix}$$

3.2 Sparse reordering for minimizing fill-in terms

To reduce the memory and time used in the factorization phase, an available reordering algorithm is called before the assembly phase to reduce fill-in terms that occur in K_{ii} matrix during the factorization phase. The Metis (Karypis, Schloegel and Kumar) reordering algorithm could be called in this step to reorder the matrix. The output from these reordering algorithms is the arrays, IPERM and INVP, mapping between the original array and the permuted array. From the Metis manual, IPERM and INVP are vectors, each of size n, where n is the number of degrees of freedom in the domain. Upon

successful execution of Metis, these 2 arrays store the fill-reducing permutation and inverse-permutation. Supposedly, $[A]$ is the original matrix, and $[A]'$ is the permuted matrix. The arrays IPERM and INVP are defined as follows.

Row or column i of $[A]'$ is the IPERM(i) row or column of $[A]$, and row or column i of $[A]$ is the INVP(i) row or column of $[A]'$. Supposedly, the output from the matrix is the following.

$$IPERM\begin{Bmatrix}1\\2\\3\\4\\5\end{Bmatrix}=\begin{Bmatrix}2\\3\\5\\4\\1\end{Bmatrix}$$

, or

IPERM(new numbering system) = {old numbering system}

Also,

$$INVP\begin{Bmatrix}1\\2\\3\\4\\5\end{Bmatrix}=\begin{Bmatrix}5\\1\\2\\4\\3\end{Bmatrix}$$

, or

INVP(old numbering system) = {new numbering system}

To demonstrate the output from Metis, one obtains that row 2 of the new reordered

matrix is row 3 of the original old matrix ($IPERM(2)=3$). Likewise, row 5 of the original old matrix is row 3 of the new reordered matrix ($INVP(5)=3$).

In order to reorder the matrix, the reordering algorithms require the adjacency information of all degrees of freedom. The adjacency information will be in the form of adjacency arrays, *IAKEEP* and *JA* . Figure 3.1 illustrates the adjacency arrays by an example on a 9 degrees of freedom system.

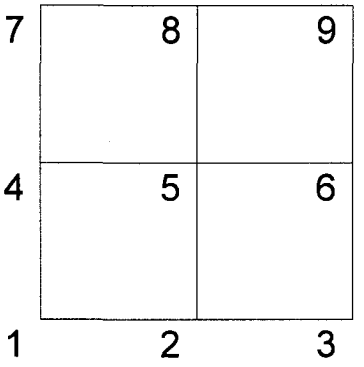


Figure 3.1: 9 degrees of freedom - rectangular element example

In Figure 3.1 example, there are 3 degrees of freedom, which are dof 2, 4 and 5, adjacent to degree of freedom 1. Likewise, there are 5 dofs, which are dof 2, 3, 5, 8 and 9, adjacent to dof 6. One could obtain IAKEEP and JA as:

$$IAKEEP = [1,4,9,12,17,25,30,33,38,41]$$

$$JA = [2,4,5,\ 1,3,4,5,6,\ 2,5,6,\ 1,2,5,7,8,\ 1,2,3,4,6,7,8,9,\ 2,3,5,8,9,\ 4,5,8,\ 4,5,6,7,9,\ 5,6,8]$$

3.3 Sparse symbolical assembly

There are 2 steps involved during the assembly phase. The first step is the symbolic

assembly phase, which will find the non-zero patterns in the stiffness matrix. The second step is the numerical assembly phase, which will find the non-zero values in the stiffness matrix.

To demonstrate the assembly processes, a 4-node, 5-element truss example is recalled in Figure 3.2. In this example, there are 4 nodes that have 2 dof on each node, so the total number of degrees of freedom is 8. Also, the elements are 2-D truss elements. Assuming Young's modulus (E) is 10 $\frac{K}{in.^2}$, and cross-sectional area is 1 $in.^2$

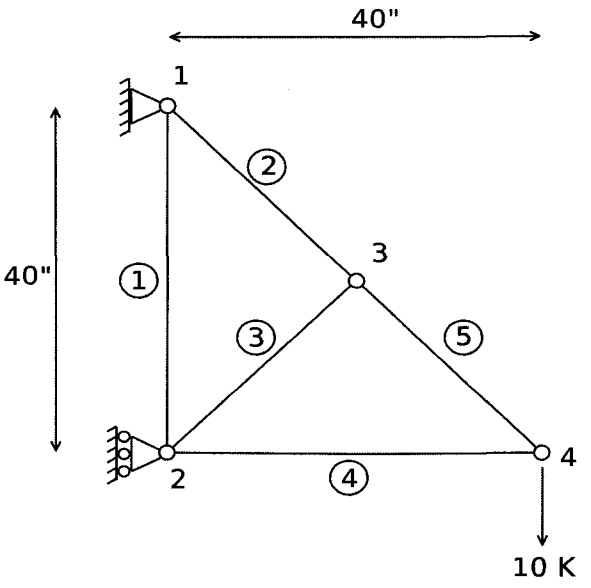


Figure 3.2: 4-node, 5-element truss example

In order to construct the non-zero pattern of the stiffness matrix, element connectivity information is required. The element connectivity information is represented by 2 integer CSR arrays, IE and JE. IE array represents the starting non-zero location of each element, and JE represents the list of actual dofs associated with the system matrix. For this particular example, IE and JE could be described as follows:

$$IE = [1,5,9,13,17,21]$$

$$JE = [1,2,3,4,1,2,5,6,3,4,5,6,3,4,7,8,5,6,7,8]$$

From the IE and JE information above, element 3 connects to 4 dofs ($IE(4) - IE(3)$), which are dof number 3, 4, 5 and 6. Then, the symbolic assembly subroutine in (Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications) is called to obtain IA and JA data. In this example, one could obtain IA and JA as,

$$IA = [1,6,10,15,19,22,24,25,25]$$

$$JA = [2,3,4,5,6,3,4,5,6,4,5,6,7,8,5,6,7,8,6,7,8,7,8,8]$$

Likewise, the total stiffness matrix will have the non-zero patterns as in Figure 3.3. The distribution of the element stiffness matrices over the total stiffness matrix is also shown in Figure 3.3. Please also note that each element in element stiffness matrices have the form $k_{ij}^{(e)}$ where i and j indicate row and column number of the element stiffness matrix and e indicates the element id in the structure.

$$K = \begin{bmatrix}
k_{1,1}^{(1)} + k_{1,1}^{(2)} & k_{1,2}^{(1)} + k_{1,2}^{(2)} & k_{1,3}^{(1)} & k_{1,4}^{(1)} & k_{1,3}^{(2)} & k_{1,4}^{(2)} & 0 & 0 \\
k_{2,1}^{(1)} + k_{2,1}^{(2)} & k_{2,2}^{(1)} + k_{2,2}^{(2)} & k_{2,3}^{(1)} & k_{2,4}^{(1)} & k_{2,3}^{(2)} & k_{2,4}^{(2)} & 0 & 0 \\
k_{3,1}^{(1)} & k_{3,2}^{(1)} & k_{3,3}^{(1)} + k_{1,1}^{(3)} + k_{1,1}^{(4)} & k_{3,4}^{(1)} + k_{1,2}^{(3)} + k_{1,2}^{(4)} & k_{1,3}^{(3)} & k_{1,4}^{(3)} & k_{1,3}^{(4)} & k_{1,4}^{(4)} \\
k_{4,1}^{(1)} & k_{4,2}^{(1)} & k_{4,3}^{(1)} + k_{2,1}^{(3)} + k_{2,1}^{(4)} & k_{4,4}^{(1)} + k_{2,2}^{(3)} + k_{2,2}^{(4)} & k_{2,3}^{(3)} & k_{2,4}^{(3)} & k_{2,3}^{(4)} & k_{2,4}^{(4)} \\
k_{3,1}^{(2)} & k_{3,2}^{(2)} & k_{3,1}^{(3)} & k_{3,2}^{(3)} & k_{3,3}^{(2)} + k_{3,3}^{(3)} + k_{1,1}^{(5)} & k_{3,4}^{(2)} + k_{3,4}^{(3)} + k_{1,2}^{(5)} & k_{1,3}^{(5)} & k_{1,4}^{(5)} \\
k_{4,1}^{(2)} & k_{4,2}^{(2)} & k_{4,1}^{(3)} & k_{4,2}^{(3)} & k_{4,3}^{(2)} + k_{4,3}^{(3)} + k_{2,1}^{(5)} & k_{4,4}^{(2)} + k_{4,4}^{(3)} + k_{2,2}^{(5)} & k_{2,3}^{(5)} & k_{2,4}^{(5)} \\
0 & 0 & k_{3,1}^{(4)} & k_{3,2}^{(4)} & k_{3,1}^{(5)} & k_{3,2}^{(5)} & k_{3,3}^{(4)} + k_{3,3}^{(5)} & k_{3,4}^{(4)} + k_{3,4}^{(5)} \\
0 & 0 & k_{4,1}^{(4)} & k_{4,2}^{(4)} & k_{4,1}^{(5)} & k_{4,2}^{(5)} & k_{4,3}^{(4)} + k_{4,3}^{(5)} & k_{4,4}^{(4)} + k_{4,4}^{(5)}
\end{bmatrix}$$

Figure 3.3: Total system stiffness matrix, K

3.4 Sparse numerical assembly

The details of the numerical assembly phase could be found in (Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications).

3.5 Sparse symbolic factorization

The symbolic factorization will be performed in order to find the locations of all non-zero, off-diagonal terms of the factorized matrix U. Also, the purpose of this phase is to find the memory required for the subsequent numerical factorization, which will be done later in the next step. The output factorized matrix U from this phase will be 2 integer CSR format arrays. The first array is IU, which will store the starting index of the first non-zero element of each row in array JU. The other array is JU(NCOEF2), which will store the (row-by-row) column index of each non-zero element in the upper triangular part of the factorized matrix. NCOEF2 is the total number of non-zero in the upper triangular part of the factorized matrix (i.e. $ncoef2 = iu(n+1)-1$). It should be noted that IU and JU will play the same roles as IA and JA.

3.6 Super-Nodes, Super-degrees of freedom (DOF)

For real life large-scale applications, after the symbolic factorization phase, several consecutive rows having the same non-zero patterns are observed. From the example matrix given in chapter 3.1, after the symbolical factorization phase, one gets the factorized matrix as follows:

$$U = \begin{bmatrix} X & 0 & X & 0 & X \\ 0 & X & 0 & 0 & X \\ X & 0 & X & X & F \\ 0 & 0 & X & X & X \\ X & X & F & X & X \end{bmatrix}$$

The symbols F in the above example represent fill-ins after factorization.

One could observe that rows 3, 4 and 5 have the same non-zero patterns. The array ISUP of size n, which is 5 in this case, is used to identify the rows having the same non-zero patterns. In this example, ISUP could be expressed as:

$$isup = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 0 \\ 0 \end{bmatrix}$$

3.7 Unrolling strategies

The super DOF information from section 3.6 and unrolling techniques described in (Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications) can be utilized effectively during the numerical factorization phase to enhance the performance of the process.

3.8 Sparse numerical (LDL^T) factorization for symmetrical matrices

The LDL^T factorization could be expressed as the following form,

$$[K] = [L][D][L]^T \tag{3.1}$$

Where

[K] is the originally matrix,

[L] is a lower triangular matrix with unit values for its diagonal, and

[D] is a diagonal matrix.

Assuming that [K] is a full matrix, the pseudo LDL^T FORTRAN code is given in the next table.

Table 3.1: Pseudo LDLT FORTRAN code

```
do I = 1,N
    do II = 1,I-1
        xmult = K(II,I)/K(II,II)
        do J = I,N
            K(I,J) = K(I,J)-xmult*K(II,J)
        enddo
        K(II,I) = xmult
    enddo
enddo
```

3.9 Sparse numerical LU factorization for unsymmetrical matrices

The LU factorization could be expressed as the following form,

$$[K]=[L][U] \tag{3.2}$$

Where

[K] is the original matrix,

[L] is the lower triangular matrix, and

[U] is the upper triangular matrix, and $[U]^T \neq [L]$

3.10 Sparse forward and backward solutions

From the system of equation,

$$[K]\cdot\vec{x} = \vec{f} \tag{3.3}$$

For the symmetrical system, one obtains

$$[L]\cdot[D]\cdot[L]^T\cdot\vec{x} = \vec{f} \tag{3.4}$$

Also, for the unsymmetrical system,

$$[L]\cdot[U]\cdot\vec{x} = \vec{f} \tag{3.5}$$

After factorized matrix information was found in the previous step, the forward solution phase will be performed to solve the equation.

$$[L]\cdot\vec{y} = \vec{f} \tag{3.6}$$

where, $\vec{y} = [D]\cdot[L]^T\cdot\vec{x}$ on the symmetrical system of equation and $\vec{y} = [U]\cdot\vec{x}$ on the unsymmetrical system of equation.

In general, y_j could be expressed as

$$y_j = \frac{f_j - \sum_{i=1}^{j-1} L_{ij}\cdot y_i}{L_{jj}} \tag{3.7}$$

After obtaining $\vec{y}$, the backward solution phase is performed.

For the symmetrical system of equation, one will solve

$$[L]^T \cdot \vec{x} = [D]^{-1} \cdot \vec{y} \tag{3.8}$$

Then, $\vec{x}$ can be expressed as

$$x_j = \frac{\frac{y_j}{D_j} - \sum_{i=j+1}^{N} L_{ji} \cdot x_i}{L_{jj}} \tag{3.9}$$

For the unsymmetrical system of equation, one will solve

$$[U] \cdot \vec{x} = \vec{y} \tag{3.10}$$

Then, $\vec{x}$ can be expressed as

$$x_j = \frac{y_j - \sum_{i=j+1}^{N} U_{ji} \cdot x_i}{U_{jj}} \tag{3.11}$$

CHAPTER IV

ITERATIVE SOLVERS

4.1 Introduction

As discussed earlier in chapter 2.2, iterative solvers are suitable to solve the system of equation in equation (2.8). By using iterative solvers, not only has the matrix $\overline{K}_B$ never been formed explicitly but also the triple product $K_{BI}^{(r)} \cdot \left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)}$ has never been computed. In addition, the efficient parallel procedures for matrix times vector can be utilized to compute $\overline{K}_B \cdot \vec{v}$. In this chapter, two iterative solvers are discussed. The first one is Preconditioned Conjugate Gradient, PCG (Hestenes and Stiefel), which is used to solve the system of the symmetrical matrix, and the other one is Flexible Generalized Minimum Residual, FGMRES(m) (Saad, "A Flexible Inner-Outer Preconditioned GMRES Algorithm"; Dongarra, Duff and Sorensen), which is used to solve the system of the unsymmetrical matrix.

4.2 Preconditioned matrix for iterative solvers with DD formulation

The main purpose of this step is to improve the condition of the stiffness matrix, $\overline{K}_B$. However, since $\overline{K}_B = \sum_{r=1}^{nsub}\left[K_{BB}^{(r)} - K_{BI}^{(r)} \cdot \left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)}\right]$ has never been explicitly assembled, an approximation of $\overline{K}_B$ has to be made. In this work, three preconditioned matrices are implemented and explained below.

Option 1: Neglect the triple product term, $K_{BI}^{(r)} \cdot \left[K_{II}^{(r)}\right]^{-1} \cdot K_{IB}^{(r)}$, and use the diagonal of K_{BB} as the preconditioned matrix.

$$[B] \approx [K_{BB}]^{-1}_{diag} \approx \left[\sum_{r=1}^{nsub} K^{(r)}_{BB,diag} \right]^{-1}$$

Option 2: Approximate $K^{(r)}_{II}$ in triple product term to be a diagonal matrix. Then,

$$[B] \approx \left[\sum_{r=1}^{nsub} \left[K^{(r)}_{BB,diag} - \left[K^{(r)}_{BI} \cdot \left[K^{(r)}_{II,diag} \right]^{-1} \cdot K^{(r)}_{IB} \right]_{diag} \right] \right]^{-1}$$

Option 3: Use GMRESM to approximate $\overline{K}_B$.

From,

$$[B] \approx \left[\overline{K}_B \right]^{-1}$$

Then, performing $[B]\vec{v}$ in iterative solver is similar to approximate $\left[\overline{K}_B \right]^{-1} \cdot \vec{v}$. Thus, one can obtain $[B]\vec{v}$ by using an iterative solver with large error tolerance setting to solve $\left[\overline{K}_B \right]^{-1} \cdot \vec{v}$. In this work, GMRES(m) is selected as an approximator for $\left[\overline{K}_B \right]^{-1} \cdot \vec{v}$ because it can handle both symmetrical and unsymmetrical problems. In addition, using this option in an iterative solver will create other iteration loops inside the main iteration loops, so the memory requirement in the solver is also a concern.

4.3 Preconditioned Conjugate Gradient (PCG)

For a symmetrical system of equation, Preconditioned Conjugate Gradient, PCG, is selected as the iterative solver in this work. The serial version of PCG is explained first. Then, the storage scheme and the parallel version of PCG are explained afterward. The

serial version of PCG algorithm is summarized in Table 4.1.

Table 4.1: Pseudo code for serial PCG algorithm

Initialized $\vec{x}_0 = \vec{0}$

Residual Vector $\vec{r}_0 = \vec{b}$ (or $\vec{r}_0 = \vec{b} - A\vec{x}_0$, for non-zero initial guess, $\vec{x}_0$)

Preconditioned step $\vec{z}_0 = [B]^{-1} \cdot \vec{r}_0$

Initial search direction $\vec{d}_0 = \vec{z}_0$

For i = 1 to maxiter

$$\alpha_i = \frac{r_i^T z_i}{d_i^T (A \cdot d_i)}$$

$$x_{i+1} = x_i + \alpha_i d_i$$

$$r_{i+1} = r_i - \alpha_i [A d_i]$$

Check for convergence; stop if $\|r_{i+1}\| < \|r_0\| \cdot \varepsilon$

$$z_{i+1} = B^{-1} r_{i+1}$$

$$\beta_i = \frac{r_{i+1}^T z_{i+1}}{r_i^T z_i}$$

$$d_{i+1} = z_{i+1} + \beta_i d_i$$

End For

4.4 Data storage scheme in parallel PCG

From the serial version of PCG algorithm, there are 8 working arrays (i.e. r_0, r_{i+1}, z_0, z_{i+1}, b, d_0, d_{i+1} and x_{i+1}) used in the code. Normally, these arrays are of size ngbjdof, total boundary degrees of freedom in the entire domain. For very large scale problems, required memory for these arrays might be large, and exceed the amount of available

memory. Therefore, partitioning of these arrays and storing them on groups of processors can help to solve large-scale problems. Basically, the processors used in the iterative solvers are divided into sub-groups. Hence, the maximum number of sub-groups is number of processors, NP, and the minimum number of sub-groups is 1. Then, the communication between processors will occur within their sub-groups. Setting number of sub-groups to be large requires more memory to store the working arrays, but the communication between processors in their group will be less. On the other hand, setting the number of sub-groups to be small requires less memory to store the working arrays, but the communication between processors in their group will be more. There are 7 parameters for this storage scheme.

1. maxnppg is the maximum number of processors per group.
2. npg is number of processors per group.
3. nog is number of groups of processors.
4. mygroup is the group ID of the processor.
5. myid is the ID of the processor in the group.
6. mysize is the partial size of ngbjdof that the processor will store.
7. MPI_COMM_WORLD2 is the MPI communicator.

For example, if 15 processors are solving a 120,000 boundary degrees of freedom problem, and maxnppg is set to be 4, the parameters on each processor are listed in Table 4.2.

Table 4.2: Parameters for data storage scheme in parallel PCG for a 120,000 dofs example

Processor ID	npg	nog	mygroup	myid	mysize	Starting location	Ending location
0	4	4	0	0	30,000	1	30,000
1	4	4	1	0	30,000	1	30,000
2	4	4	2	0	30,000	1	30,000
3	3	4	3	0	40,000	1	40,000
4	4	4	0	1	30,000	30,001	60,000
5	4	4	1	1	30,000	30,001	60,000
6	4	4	2	1	30,000	30,001	60,000
7	3	4	3	1	40,000	40,001	80,000
8	4	4	0	2	30,000	60,001	90,000
9	4	4	1	2	30,000	60,001	90,000
10	4	4	2	2	30,000	60,001	90,000
11	3	4	3	2	40,000	80,001	120,000
12	4	4	0	3	30,000	90,001	120,000
13	4	4	1	3	30,000	90,001	120,000
14	4	4	2	3	30,000	90,001	120,000

From Table 4.2, starting and ending locations indicate the range of index in the working arrays stored on the processor. Beside, MPI_COMM_WORLD2 parameter is obtained by calling MPI_COMM_SPLIT subroutine with mygroup and myid as color and key input parameters, respectively. Finally, the working arrays from serial version of PCG are reorganized and listed in Table 4.3.

Table 4.3: Name and size of working arrays used in parallel PCG

Working array	Name in the code	Size
COMB	COMB	ngbjdof
r_i	RI	mysize
r_{i+1}	RIP1	mysize
B	B	mysize
z_i	ZI	mysize
z_{i+1}	ZIP1	mysize
d_i	DI	mysize
STORED	STORED	mysize
z_{bi}	ZBIPART	mysize

The parallel version of PCG, therefore, can be summarized in Table 4.4.

Table 4.4: Pseudo-parallel PCG algorithm

Step 1: Assign sub-group information to each processor

Step 2: Partition initial guess (x_b) and store in ZBIPART array

Step 3: Compute $\overline{F}_B = \sum_{r=1}^{nsub} \overline{F}_B^{(r)}$ using the procedures discussed in chapter 2.3 and store the result in COMB array, which, in this case, represents $\overline{\overline{F}}_B$.

Step 4: Compute $\overline{K}_B \cdot x_0 = \sum_{r=1}^{nsub} \overline{K}_B^{(r)} \cdot x_0$ using the procedures discussed in chapter 2.3 and store the result in TEMP2 array.

Step 5: Compute the residual

$$\vec{r_0} = \overline{F}_B - \overline{K}_B \cdot x_0 = COMB - TEMP2$$

Step 6: Check for the convergence. Exit if

$$\left\| \vec{r_0} \cdot \vec{r_0} \right\| < \left\| \overline{F}_B \cdot \overline{F}_B \right\| \cdot \varepsilon$$

Step 7: Construct preconditioned matrix $[B]$

Step 8: Compute $\overrightarrow{z_0^{(myid)}} = \left[B^{(myid)}\right]^{-1} \cdot \overrightarrow{r_0^{(myid)}}$

Step 9: $d_0^{(myid)} = r_0^{(myid)}$

Begin Iteration loops (from 1,2,...,maxiter)

Step 10: Combine $d_i^{(myid)}$ array from all processors within the sub-group

$$COMB = \sum_{myid=0}^{npg-1} d_i^{(myid)}$$

Step 11: Compute $\overline{K}_B \cdot d_i^{combined}$ using the procedures discussed in chapter 2.3

$$COMB = \overline{K}_B \cdot d_i^{combined}$$

Step 12: Compute the step size

$$\alpha_i = \sum_{myid=0}^{npg-1} \alpha_i^{(myid)}$$

where

$$\alpha_i^{(myid)} = \frac{r_i^{(myid)} \cdot z_i^{(myid)}}{d_i^{(myid)} \cdot COMB^{(myid)}}$$

Step 13: Compute new, improved solution

$$z_{bi}^{(myid)} = z_{bi}^{(myid)} + \alpha_i \cdot d_i^{(myid)}$$

Step 14: Compute new residual vector

$$r_{i+1}^{(myid)} = r_i^{(myid)} - \alpha_i \cdot \left[\overline{K}_B \cdot d_i^{combined} \right] = r_i^{(myid)} - \alpha_i \cdot COMB^{(myid)}$$

Step 15: Check for convergence. Exit the loop if

$$\left\| \overrightarrow{r_{i+1}} \cdot \overrightarrow{r_{i+1}} \right\| < \left\| \overline{F}_B \cdot \overline{F}_B \right\| \cdot \varepsilon$$

where

$$\overrightarrow{r_{i+1}} = \sum_{myid=0}^{npg-1} r_{i+1}^{(myid)}$$

Step 16: Preconditioning step

$$\overrightarrow{z_{i+1}^{(myid)}} = \left[B^{(myid)} \right]^{-1} \cdot \overrightarrow{r_{i+1}^{(myid)}}$$

Step 17: $\beta_i = \dfrac{\sum_{myid=0}^{npg-1}\left[r_{i+1}^{(myid)} \cdot z_{i+1}^{(myid)}\right]}{\sum_{myid=0}^{npg-1}\left[r_i^{(myid)} \cdot z_i^{(myid)}\right]}$

Step 18: $d_i^{(myid)} = z_{i+1}^{(myid)} + \beta_i \cdot d_i^{(myid)}$

Step 19: $r_i^{(myid)} = r_{i+1}^{(myid)}$

Step 20: $z_i^{(myid)} = z_{i+1}^{(myid)}$

End Iteration loops

Step 21: Combine $z_{bi}^{(myid)}$ array from all processors within the sub-group

$$x_b = \sum_{myid=0}^{npg-1} z_{bi}^{(myid)}$$

Exit the code with x_b as the output

4.5 Flexible Generalized Minimum Residual (FGMRES)

In this work, FGMRES(m) algorithm is selected to solve the system of unsymmetrical matrices. Unlike GMRES(m) algorithm, the preconditioned matrix in FGMRES(m) algorithm changes per iterations step. GMRES(m) is explained first in Table 4.5. Then, FGMRES(m) in Domain Decomposition context will be discussed later in Table 4.6.

Table 4.5: Pseudo-sequential version of GMRES(m)

Step 1: Initialized $x = x^0$

Step 2: Calculate initial residual

$$r = [B]^{-1}[b - [A]x^0]$$

Begin Outer Iteration loops (from j = 1,2, ... until number of inner iteration counts reaches MAXITER)

Step 3: $\beta = \|r\|_2$

Step 4: $v^1 = \frac{r}{\beta}$

Step 5: $\hat{b} = \beta e^1$

Begin Inner Iteration loops (from i = 1,2,..., m)

Step 6: $w = [B]^{-1} A v^i$

For k = 1,...,i

Step 7: $h_{k,i} = v^k \cdot w$

Step 8: $w = w - h_{k,i} v^k$

End k

Step 9: $h_{i+1,i} = \|w\|_2$

Step 10: $v^{i+1} = \frac{w}{h_{i+1,i}}$

For k = 2,...,i

Step 11: $temp = c_{k-1} h_{k-1,i} + s_{k-1} h_{k,i}$

$h_{k,i} = s_{k-1} h_{k-1,i} - c_{k-1} h_{k,i}$

$h_{k-1,i} = temp$

End k

Step 12: Compute the Givens rotation matrix parameters

if ($h_{i+1,i} = 0.0$) then

$$c_i = 1.0$$

$$s_i = 0.0$$

elseif ($|h_{i+1,i}| > |h_{i,i}|$) then

$$temp = \frac{h_{i,i}}{h_{i+1,i}}$$

$$s_i = \frac{1.0}{\sqrt{1.0 + temp^2}}$$

$$c_i = temp \cdot s_i$$

elseif ($|h_{i,i}| > |h_{i+1,i}|$) then

$$temp = \frac{h_{i+1,i}}{h_{i,i}}$$

$$c_i = \frac{1.0}{\sqrt{1.0 + temp^2}}$$

$$s_i = temp \cdot c_i$$

endif

Step 13: $h_{i,i} = c_i h_{i,i} + s_i h_{i+1,i}$

Step 14: $h_{i+1,i} = 0.0$

Step 15: $temp = c_i \hat{b}_i$

$$\hat{b}_{i+1} = -s_i \hat{b}_i$$

$$\hat{b}_i = temp$$

Step 16: $\rho = \left|\hat{b}_{i+1}\right|$

Step 17: Convergence check: If ($\rho < \varepsilon$),

Step 18: $n_r = i$; Go to SOL

End Inner Iteration loops

Step 19: $n_r = m$

SOL;

Step 20: $y_{n_r} = \dfrac{\hat{b}_{n_r}}{h_{n_r,n_r}}$

For $k = n_r - 1,..,1$

Step 21: $y_k = \dfrac{\hat{b}_k - \sum_{i=k+1}^{n_r} h_{k,i} y_i}{h_{k,k}}$

End k

Step 22: $x = x + [v]y$

Step 23: Convergence Check: Exit if ($\rho < \varepsilon$)

Step 24: $r = [B]^{-1}[b - Ax]$

End Outer Iteration loops

In step 6 of the serial GMRES(m) algorithm, the fixed preconditioning matrix $[B]$ is used to improve the condition of the coefficient matrix, $[A]$. On the other hand, in FGMRES(m), the approximation of $[B]v$ is obtained by solving $z \approx A^{-1}v$ with a large (10^{-2} to 10^{-1}) error tolerance. Thus, the preconditioner used in FGMRES(m) will change per iteration step, and FGMRES(m) algorithm in Domain Decomposition context is summarized in Table 4.6.

Table 4.6: Pseudo parallel FGMRES(m) algorithm with GMRESM(m) as a preconditioner

Step 1: Initialized $x = x^0$

Step 2: Calculate initial residual

$$r = \overline{F}_B - \overline{K}_B x^0$$

Begin Outer Iteration loops (from j = 1,2, ... until number of inner iteration counts reaches MAXITER)

Step 3: $\beta = \|r\|_2$

Step 4: $v^1 = \frac{r}{\beta}$

Step 5: $\hat{b} = \beta e^1$ where e^1 is the first unit vector

Begin Inner Iteration loops (from i = 1,2,..., m)

Step 6: Compute $z^i \approx \overline{K}_B^{-1} v^i$ using GMRESM for approximation

Step 7: $w = \overline{K}_B z^i$

For k = 1,...,i

Step 8: $h_{k,i} = v^k \cdot w$

Step 9: $w = w - h_{k,i} v^k$

End k

Step 10: $h_{i+1,i} = \|w\|_2$

Step 11: $v^{i+1} = \frac{w}{h_{i+1,i}}$

For k = 2,...,i

Step 12: $temp = c_{k-1} h_{k-1,i} + s_{k-1} h_{k,i}$

$$h_{k,i} = s_{k-1} h_{k-1,i} - c_{k-1} h_{k,i}$$

$$h_{k-1,i} = temp$$

End k

Step 13: Compute the Givens rotation matrix parameters

if ($h_{i+1,i} = 0.0$) then

$$c_i = 1.0$$

$$s_i = 0.0$$

elseif ($|h_{i+1,i}| > |h_{i,i}|$) then

$$temp = \frac{h_{i,i}}{h_{i+1,i}}$$

$$s_i = \frac{1.0}{\sqrt{1.0 + temp^2}}$$

$$c_i = temp \cdot s_i$$

elseif ($|h_{i,i}| > |h_{i+1,i}|$) then

$$temp = \frac{h_{i+1,i}}{h_{i,i}}$$

$$c_i = \frac{1.0}{\sqrt{1.0 + temp^2}}$$

$$s_i = temp \cdot c_i$$

endif

Step 14: $h_{i,i} = c_i h_{i,i} + s_i h_{i+1,i}$

Step 15: $h_{i+1,i} = 0.0$

Step 16: $\hat{b}_{i+1} = -s_i \hat{b}_i$

$$\hat{b}_i = c_i \hat{b}_i$$

Step 17: $\rho = \left| \hat{b}_{i+1} \right|$

Step 18: Convergence check: If ($\rho < \varepsilon$),

Step 19: $n_r = i$; Go to SOL

End Inner Iteration loops

Step 20: $n_r = m$

SOL;

Step 21: $y_{n_r} = \dfrac{\hat{b}_{n_r}}{h_{n_r,n_r}}$

For $k = n_r - 1,..,1$

Step 22: $y_k = \dfrac{\hat{b}_k - \sum_{i=k+1}^{n_r} h_{k,i} y_i}{h_{k,k}}$

End k

Step 23: $x = x + [z]y$

Step 24: Convergence Check: Exit if ($\rho < \varepsilon$)

Step 25: $r = \overline{F}_B - \overline{K}_B x$

End Outer Iteration loops

CHAPTER V

NUMERICAL APPLICATIONS

Software based on parallel primal Domain Decomposition formulation illustrated in this work has been developed. The software has capabilities to solve both symmetrical and unsymmetrical systems. Moreover, not only does the package have options to handle both real and complex in double precision (i.e. 64-bit arithmetic), it is also highly portable thanks to the message passing interface (MPI), which is widely available on supercomputer clusters nowadays. Results from two examples executed on Wilbur Cluster are observed and documented in this chapter.

5.1 Example 1 – Three dimensional acoustic finite element model without flow

In this example, the developed parallel DD code is exercised to study the propagation of plane acoustic pressure waves in a 3-D hard wall duct without end reflection and airflow. The duct is shown in Figure 5.1 and is modeled with brick elements. The source and exit planes are located at the left and right boundaries, respectively. The matrix, K, contains complex coefficients, and the dimension of K is determined by the product of NN, MM and QQ ($N = MMxNNxQQ$). Results are presented for 4 grids (N = 1.0, 2.5, 3.96 and 10 million degrees of freedom) and the finite element analysis procedure for generation of the complex stiffness matrix, K, was presented in (Watson, "Three-Dimensional Rectangular Duct Code With Application to Impedance Eduction").

The results are obtained from the configurations below.

The example is tested on the ODU Wilbur Cluster, which has 64 nodes, and each node has 2 processors and 4GB of memory.

Two schemes to partition the domain are used on a 3.96 million degrees of freedom example to compare the results. The first scheme is to use ParMETIS (see chapter 6.2 for details). The second scheme is to divide the domain along the Z axis and let each processor handle each piece of the partitioned subdomains.

During the symmetrical iterative solver phase on 3.96 million degrees of freedom, maximum number of processors in the group, MAXNPPG (see chapter 4.3 for details), is set to be 1, 8, 16 and 32 in order to compare the results between two data storage schemes in the symmetrical iterative solver (PCG).

From the results presented in Tables 5.1 to 5.4 and Figures 5.2 to 5.9, there are several elements that are worth mentioning.

Most computational time occurs in the factorization phase and the boundary displacements solving phase.

Partitioning time tends to increase when more processors are used. Although most of the steps explained in chapter 6.3 are running independently on each processor, the first step is running sequentially and has data communication among processors. Therefore, increasing the number of processors creates more communication time to the total time of the phase.

Reordering, assembly, factorization and interior degrees of freedom solving times decrease when the number of processors increases. The reason is that the size of subdomains will be smaller when the domain is partitioned into more subdomains.

It is clearly seen that increasing the number of processors will drastically reduce the total time and lead to the super linear speedup. This is because the size of the subdomain

stiffness matrix is much smaller compared to the entire coefficient stiffness matrix when the entire domain is divided into several small subdomains in multi-processor runs. Moreover, since the operations required to factorize a matrix of size N are proportional to $\left(N \cdot BW^2\right)$, where BW represents the half bandwidth of the matrix, for sparse matrix, the operations will drastically reduce when solving the problem on many processors.

Timing in the boundary degrees of freedom solving phase depends on the number of iterations used during the phase. In contrast, an increasing number of iterations used during the boundary degrees of freedom solving phase is not necessarily influenced by an increase of processors used.

Although the factorization time is significantly reduced when using more processors, the number of boundary degrees of freedom will also increase. This is because increasing the number of subdomains will introduce more edge cuts to the domain. As a result, the boundary degrees of freedom solving time using the iterative solver will also increase.

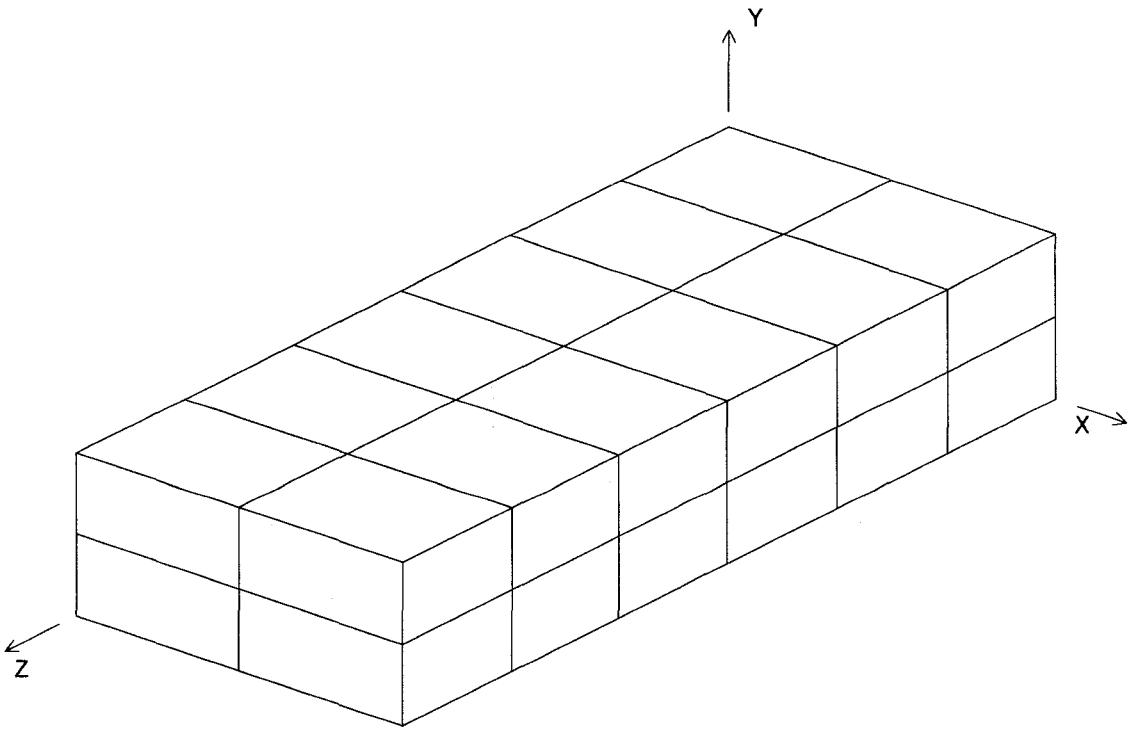


Figure 5.1: 3-D symmetrical acoustic example

Table 5.1: Timing Statistic for 1 million dofs, 3-D symmetrical acoustic example

1M	10	20	30	40	50	60	70	80	90	100
TT	335	123	74	67	64	68	77	84	95	102
ISUP	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
ASUP	1.00	2.72	4.53	5.00	5.23	4.93	4.35	3.99	3.53	3.28
MaxMem	935	399	230	166	124	98	91	76	70	63
TOT_BDOF	11664	24624	37584	50544	63504	76464	89424	102384	115344	128304
MAX_BDOF	2592	2592	2592	2592	2592	2592	2592	2592	2592	2592
MAX_IDOF	99792	49248	32400	24624	19440	15552	14256	11664	10368	9072
PT	2.1	3.5	4.8	6.2	7.6	8.9	10.3	12.1	13.7	15.2
RT	2.2	1	0.6	0.4	0.4	0.3	0.2	0.2	0.2	0.1
AT	0.8	0.5	0.3	0.3	0.2	0.2	0.2	0.2	0.3	0.2
FT	297	88	32	18	10	5	4	2	2	1
BT	31	29	35	42	46	53	61	68	78	85
NIT	32	60	108	127	142	157	170	182	193	204
IT	0.9	0.3	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

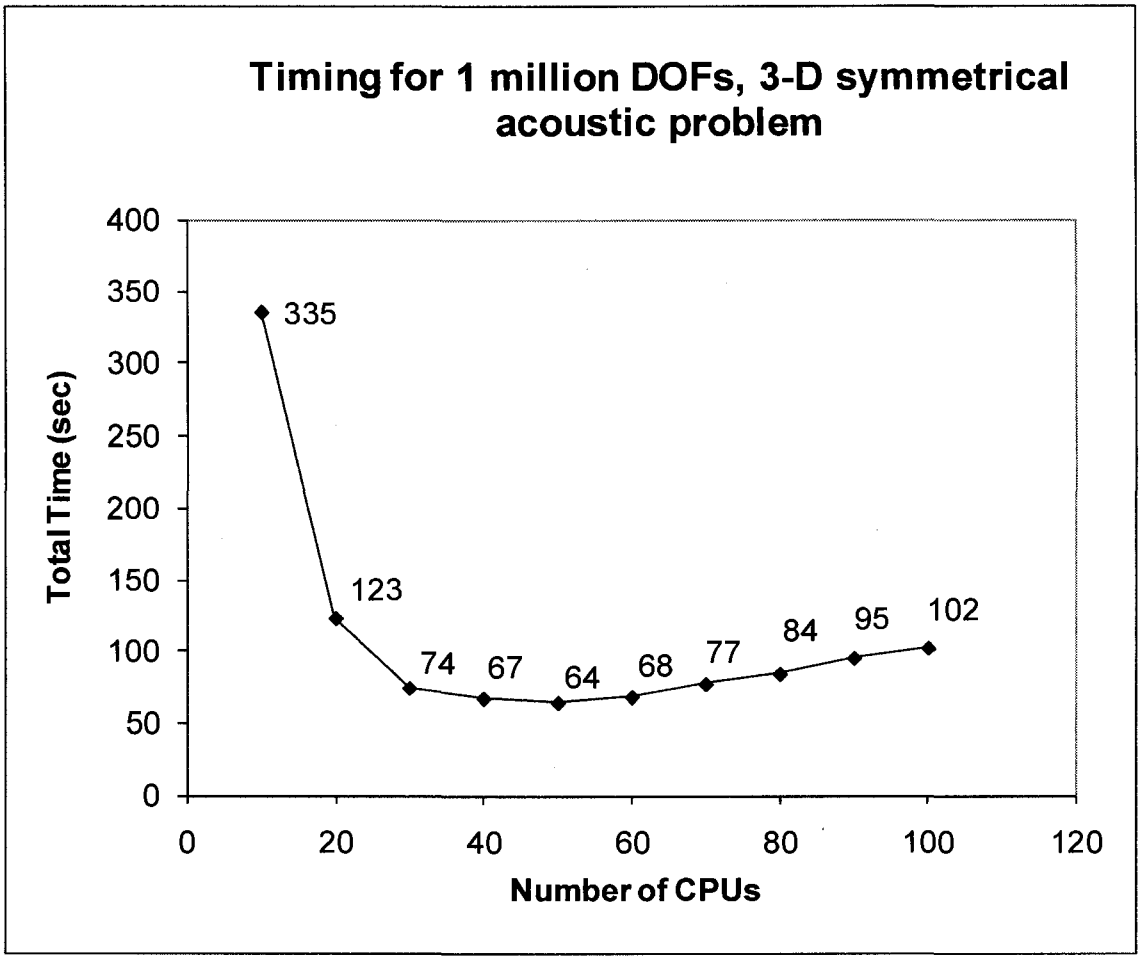


Figure 5.2: Timing for 1 million dofs, 3-D symmetrical acoustic example

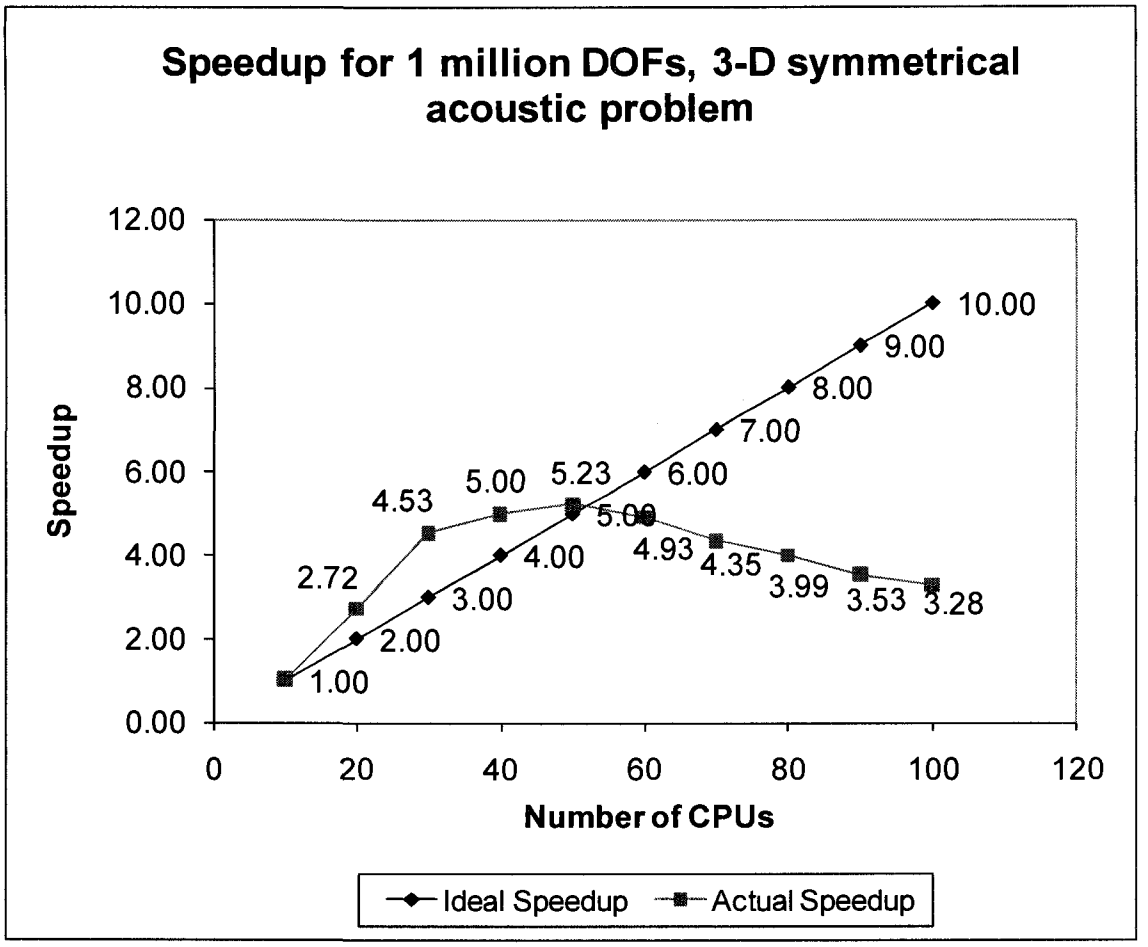


Figure 5.3: Speedup for 1 million dofs, 3-D symmetrical acoustic example

Table 5.2: Timing statistic for 2.5 million dofs, 3-D symmetrical acoustic example

2.5M	10	20	30	40	50	60	70	80	90	100
TT	-	694	353	262	211	197	202	129	229	244
ISUP	-	1.00	1.50	2.00	2.50	3.00	3.50	4.00	4.50	5.00
ASUP	-	1.00	1.97	2.65	3.29	3.52	3.44	5.38	3.03	2.84
MaxMem	-	1402	787	560	413	314	272	233	215	195
TOT_BDOF	-	47500	72500	97500	122500	147500	172500	197500	222500	247500
MAX_BDOF	-	5000	5000	5000	5000	5000	5000	5000	5000	5000
MAX_IDOF	-	125000	82500	62500	50000	40000	35000	30000	27500	25000
PT	-	7.9	10.7	13.4	16.3	19.2	21.9	25	27.5	31.1
RT	-	3	1.9	1.4	1	0.8	0.7	0.6	0.5	0.4
AT	-	1.1	2.4	0.7	0.7	2.2	1.7	0.6	0.6	0.5
FT	-	562	209	117	64	36	25	17	14	10
BT	-	117	126	128	127	137	152	84	185	200
NIT	-	71	122	148	166	185	200	104	228	241
IT	-	1.1	0.9	0.5	0.5	0.4	0.3	0.3	0.3	0.2

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

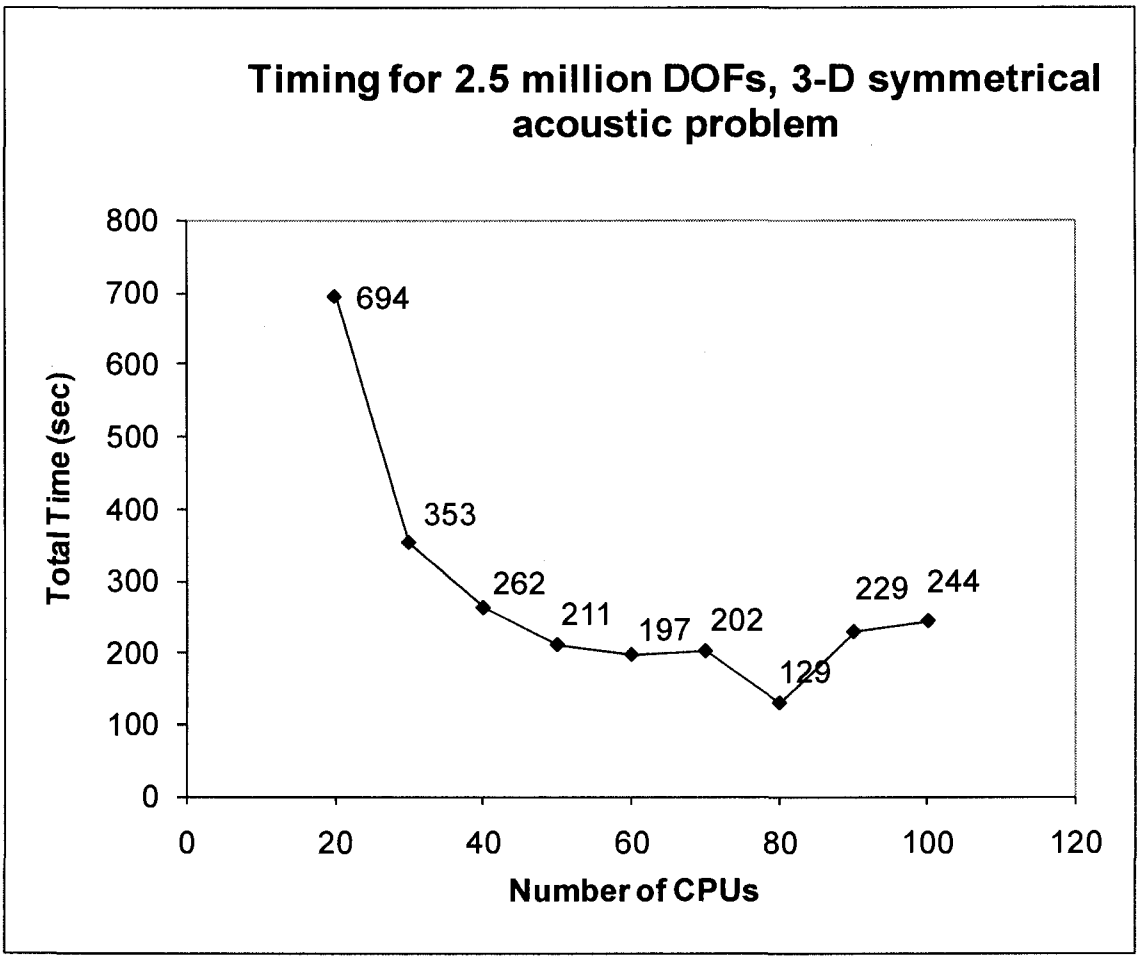


Figure 5.4: Timing for 2.5 million dofs, 3-D symmetrical acoustic example

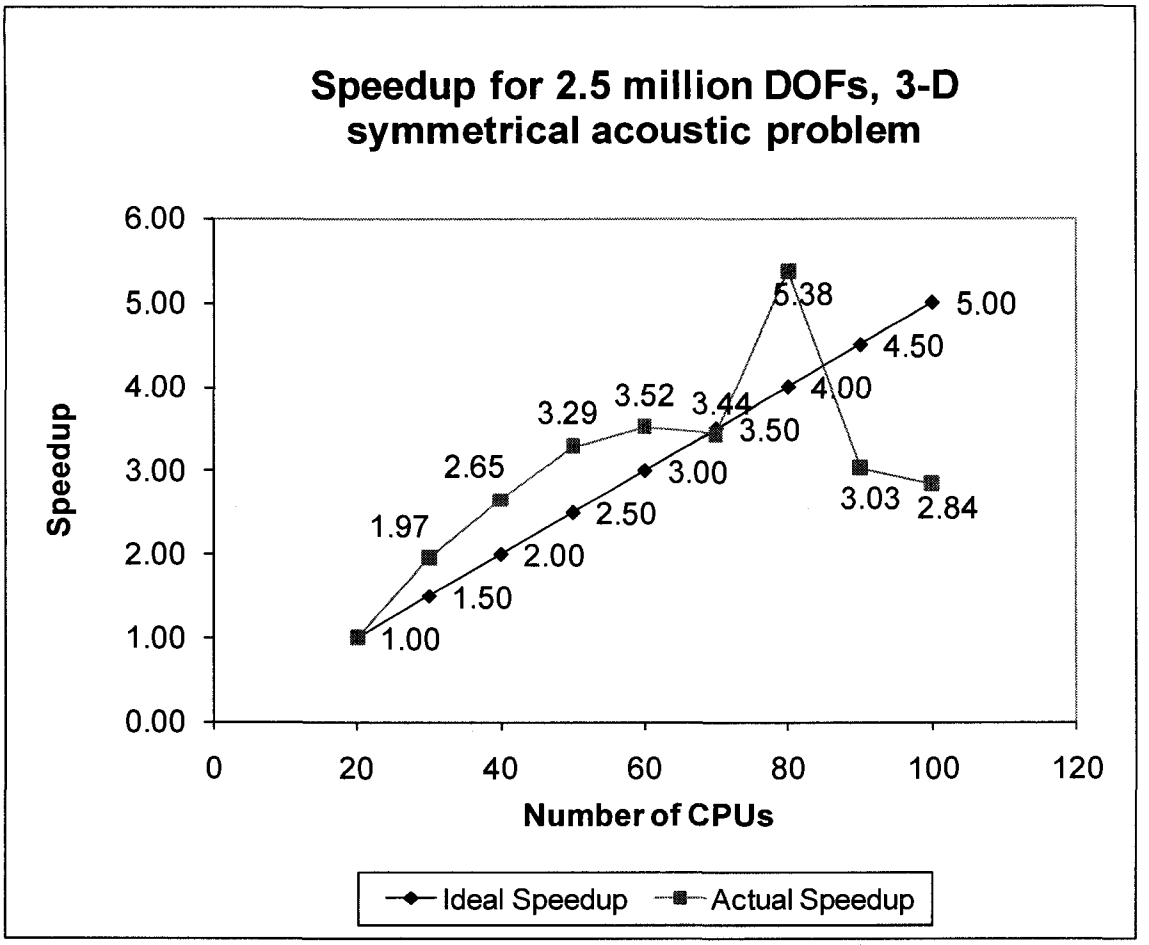


Figure 5.5: Speedup for 2.5 million dofs, 3-D symmetrical acoustic example

Table 5.3: Timing statistic for 3.96 million dofs, 3-D symmetrical acoustic example

3.96M (MAXNPPG=32)	10	20	30	40	50	60	70	80	90	100
TT	-	-	807	541	414	362	283	354	380	399
ISUP	-	-	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33
ASUP	-	-	1.00	1.49	1.95	2.23	2.85	2.28	2.12	2.02
MaxMem	-	-	1441	981	733	572	461	393	361	330
TOT_BDOF	-	-	104400	140400	176400	212400	248400	284400	320400	356400
MAX_BDOF	-	-	7200	7200	7200	7200	7200	7200	7200	7200
MAX_IDOF	-	-	129600	97200	79200	64800	54000	46800	43200	39600
PT	-	-	16.2	20.2	24.5	28.7	32.6	37.1	41.3	45.7
RT	-	-	3.2	2.2	1.8	1.4	1.1	1	0.9	0.8
AT	-	-	2.4	4.3	1.2	1.2	2.7	4.5	1	0.8
FT	-	-	531	269	151	91	56	36	30	24
BT	-	-	251	242	233	237	187	273	303	325
NIT	-	-	136	163	186	201	159	236	252	266
IT	-	-	1.2	1.1	0.8	0.6	0.6	0.5	0.5	0.2

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

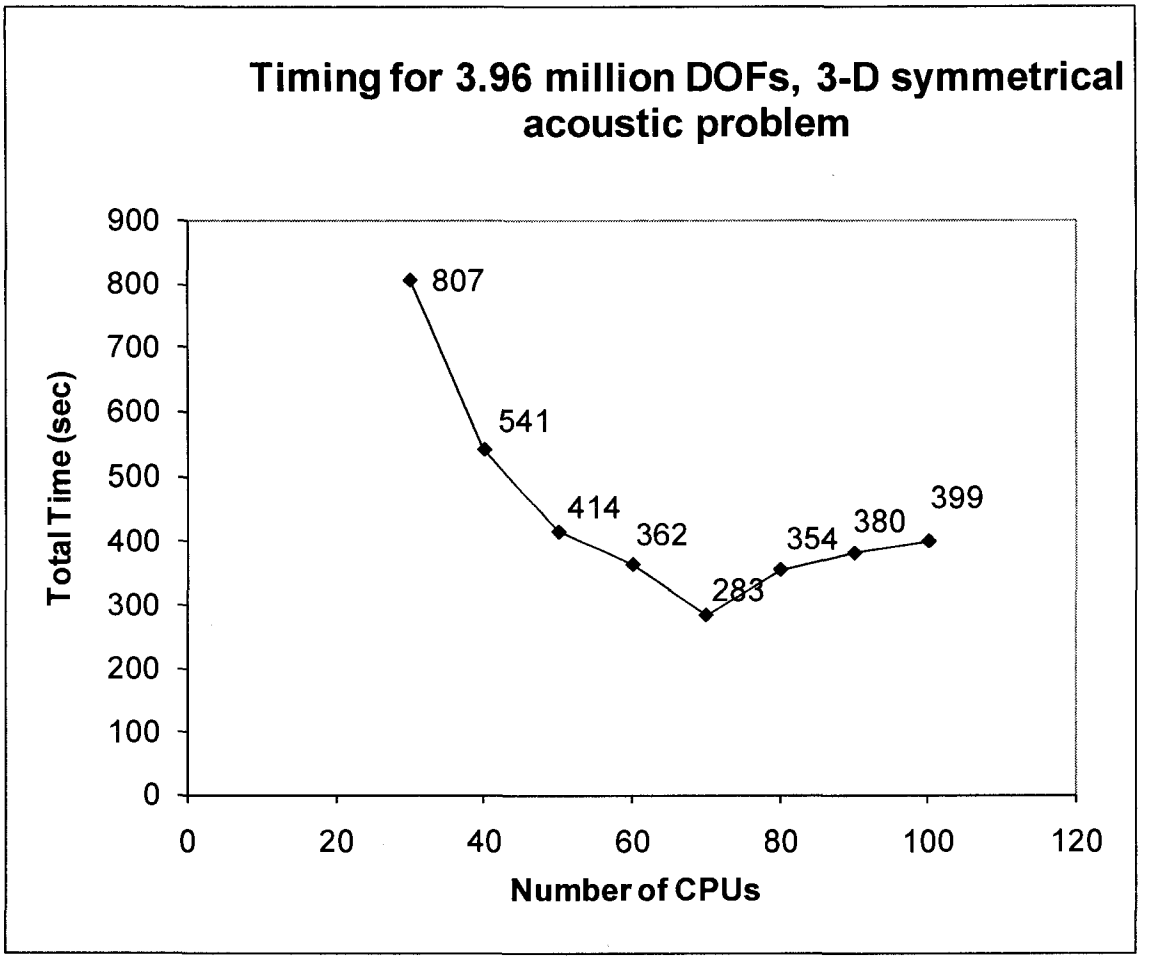


Figure 5.6: Timing for 3.96 million dofs, 3-D symmetrical acoustic example

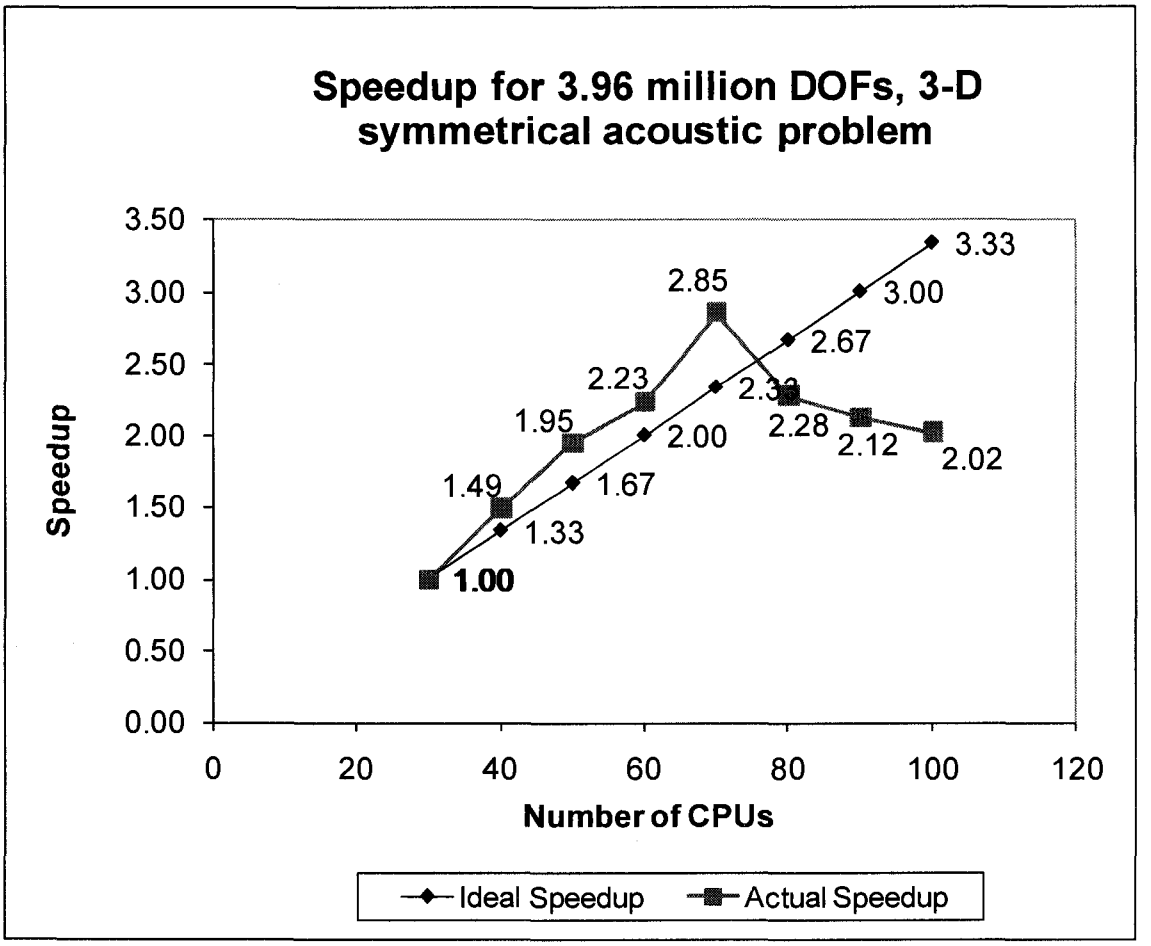


Figure 5.7: Speedup for 3.96 million dofs, 3-D symmetrical acoustic example

Table 5.4: Timing statistic for 10 million dofs, 3-D symmetrical acoustic example

10 M	10	20	30	40	50	60	70	80	90	100
TT	-	-	-	-	-	1103	877	321	313	312
ISUP	-	-	-	-	-	1.00	1.17	1.33	1.50	1.67
ASUP	-	-	-	-	-	1.00	1.26	3.44	3.52	3.54
MaxMem	-	-	-	-	-	1611	1357	1133	1034	920
TOT_BDOF	-	-	-	-	-	590000	690000	790000	890000	990000
MAX_BDOF	-	-	-	-	-	20000	20000	20000	20000	20000
MAX_IDOF	-	-	-	-	-	160000	140000	120000	110000	100000
PT	-	-	-	-	-	67.6	77.3	87.2	96.9	107
RT	-	-	-	-	-	4	3.6	2.9	2.6	2.3
AT	-	-	-	-	-	18.5	33	7.5	12.9	26.5
FT	-	-	-	-	-	372	252	164	132	93
BT	-	-	-	-	-	633	504	53	61	70
NIT	-	-	-	-	-	184	159	7	7	7
IT	-	-	-	-	-	1.7	1.7	1.4	1.4	0.9

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

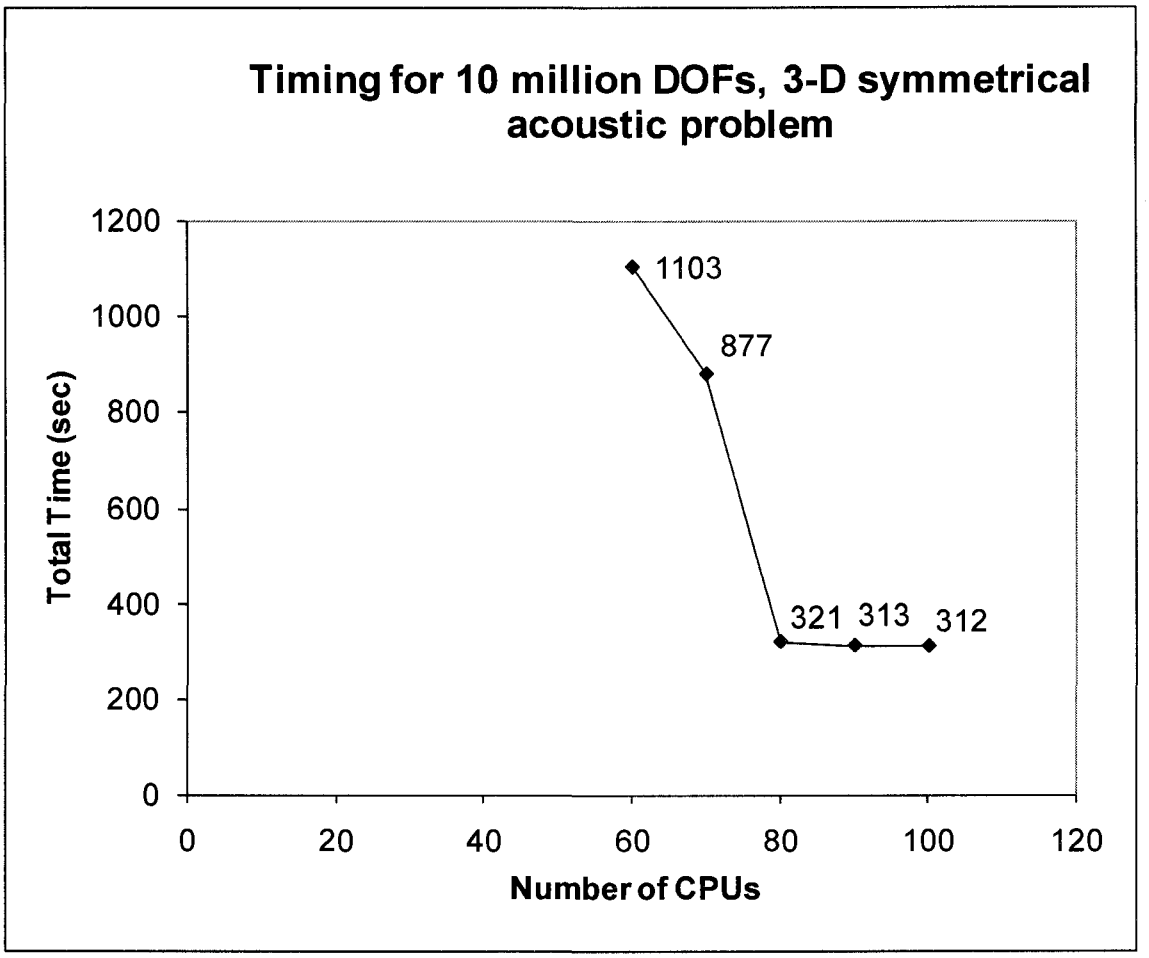


Figure 5.8: Timing for 10 million dofs, 3-D symmetrical acoustic example

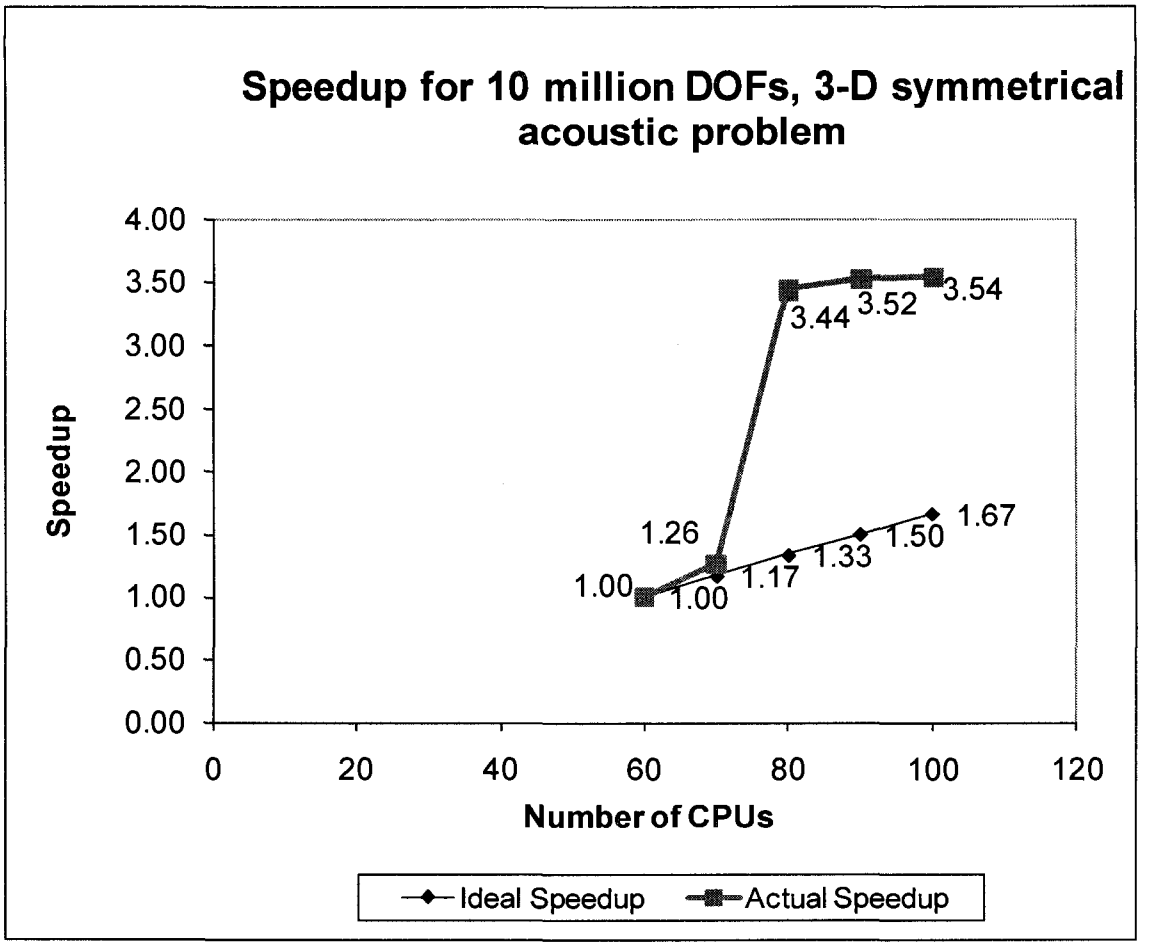


Figure 5.9: Speedup for 10 million dofs, 3-D symmetrical acoustic example

Table 5.5: Timing statistic for 3.96 million dofs (MAXNPPG=1), 3-D symmetrical acoustic example

3.96 M (MAXNPPG=1)	10	20	30	40	50	60	70	80	90	100
TT	-	-	899	604	420	324	250	259	279	274
ISUP	-	-	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33
ASUP	-	-	1.00	1.49	2.14	2.77	3.60	3.47	3.22	3.28
MaxMem	-	-	1453	998	755	598	491	428	401	373
TOT_BDOF	-	-	104400	140400	176400	212400	248400	284400	320400	356400
MAX_BDOF	-	-	7200	7200	7200	7200	7200	7200	7200	7200
MAX_IDOF	-	-	129600	97200	79200	64800	54000	46800	43200	39600
PT	-	-	16.1	20.3	24.3	28.7	32.7	37.4	41.7	45.6
RT	-	-	3.1	2.2	2.1	1.4	1.1	1	0.9	0.8
AT	-	-	3.1	4.2	1.3	1.2	2.7	4.5	1.3	0.9
FT	-	-	552	344	176	92	57	39	30	24
BT	-	-	321	230	214	198	153	175	203	200
NIT	-	-	136	163	186	201	159	236	252	266
IT	-	-	1.6	1	0.9	0.6	0.4	0.3	0.3	0.2

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

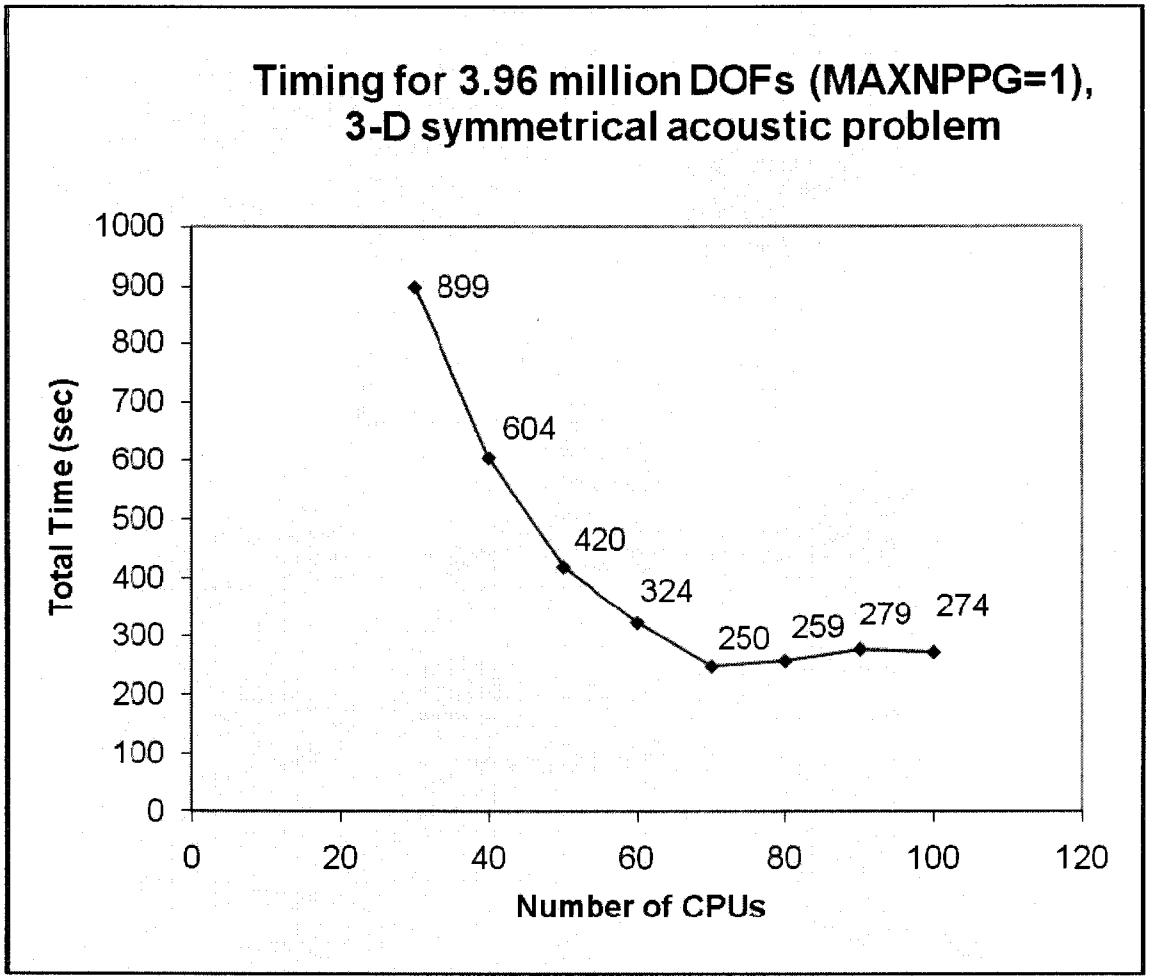


Figure 5.10: Timing for 3.96 million dofs (MAXNPPG=1), 3-D symmetrical acoustic example

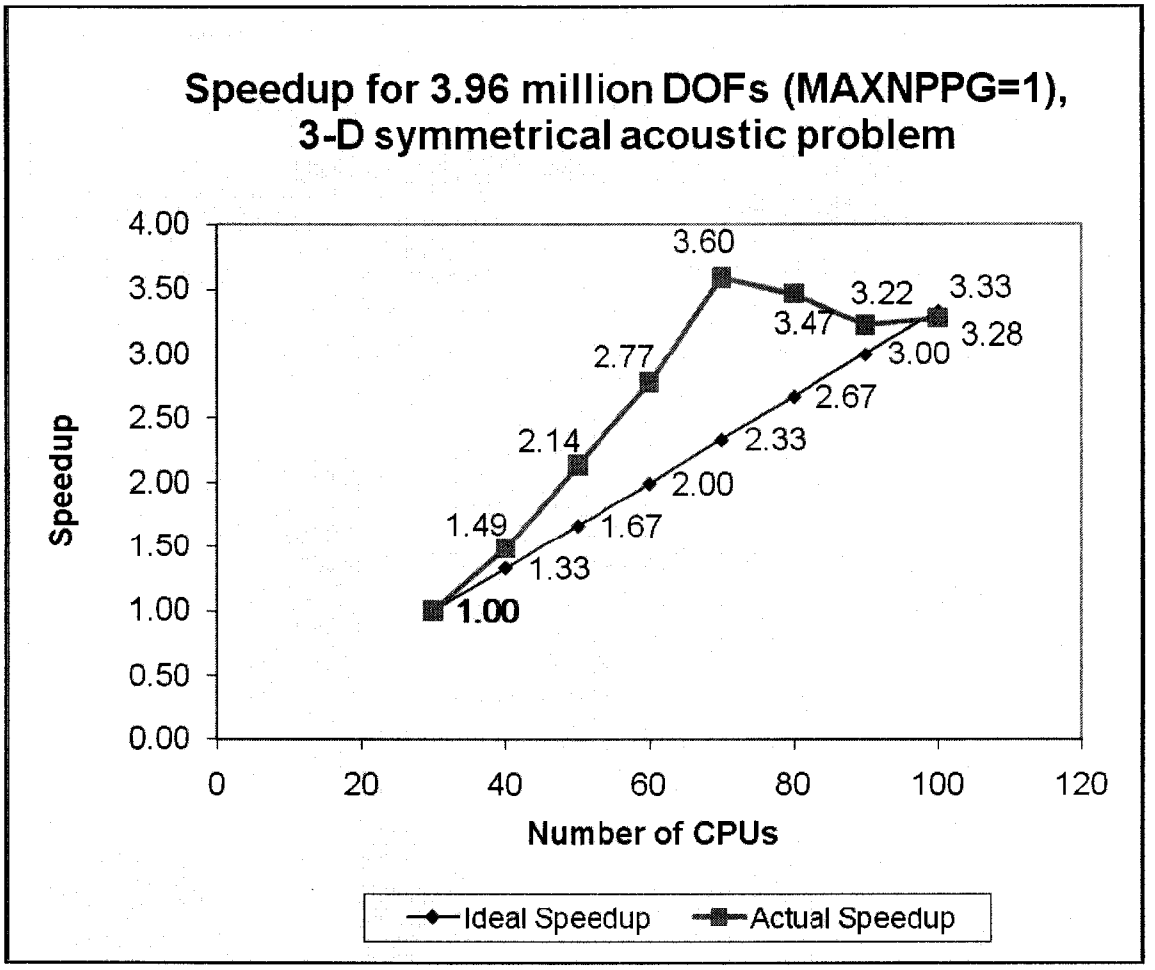


Figure 5.11: Speedup for 3.96 million dofs (MAXNPPG=1), 3-D symmetrical acoustic example

Table 5.6: Timing statistic for 3.96 million dofs (MAXNPPG=8), 3-D symmetrical acoustic example

3.96M (MAXNPPG=8)	10	20	30	40	50	60	70	80	90	100
TT	-	-	921	631	453	349	283	319	351	376
ISUP	-	-	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33
ASUP	-	-	1.00	1.46	2.03	2.64	3.25	2.89	2.62	2.45
MaxMem	-	-	1442	981	735	575	463	396	365	334
TOT_BDOF	-	-	104400	140400	176400	212400	248400	284400	320400	356400
MAX_BDOF	-	-	7200	7200	7200	7200	7200	7200	7200	7200
MAX_IDOF	-	-	129600	97200	79200	64800	54000	46800	43200	39600
PT	-	-	16.3	20.2	24.9	28.6	32.8	37.3	41.7	46
RT	-	-	3.1	2.3	1.8	1.4	1.1	0.9	0.9	0.8
AT	-	-	2.4	4.1	1.2	1.1	2.7	4.4	1.1	0.8
FT	-	-	534	269	152	92	56	38	30	23
BT	-	-	361	332	271	223	187	236	275	302
NIT	-	-	136	163	186	201	159	236	252	266
IT	-	-	1.7	0.8	0.7	0.7	0.6	0.3	0.5	0.4

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

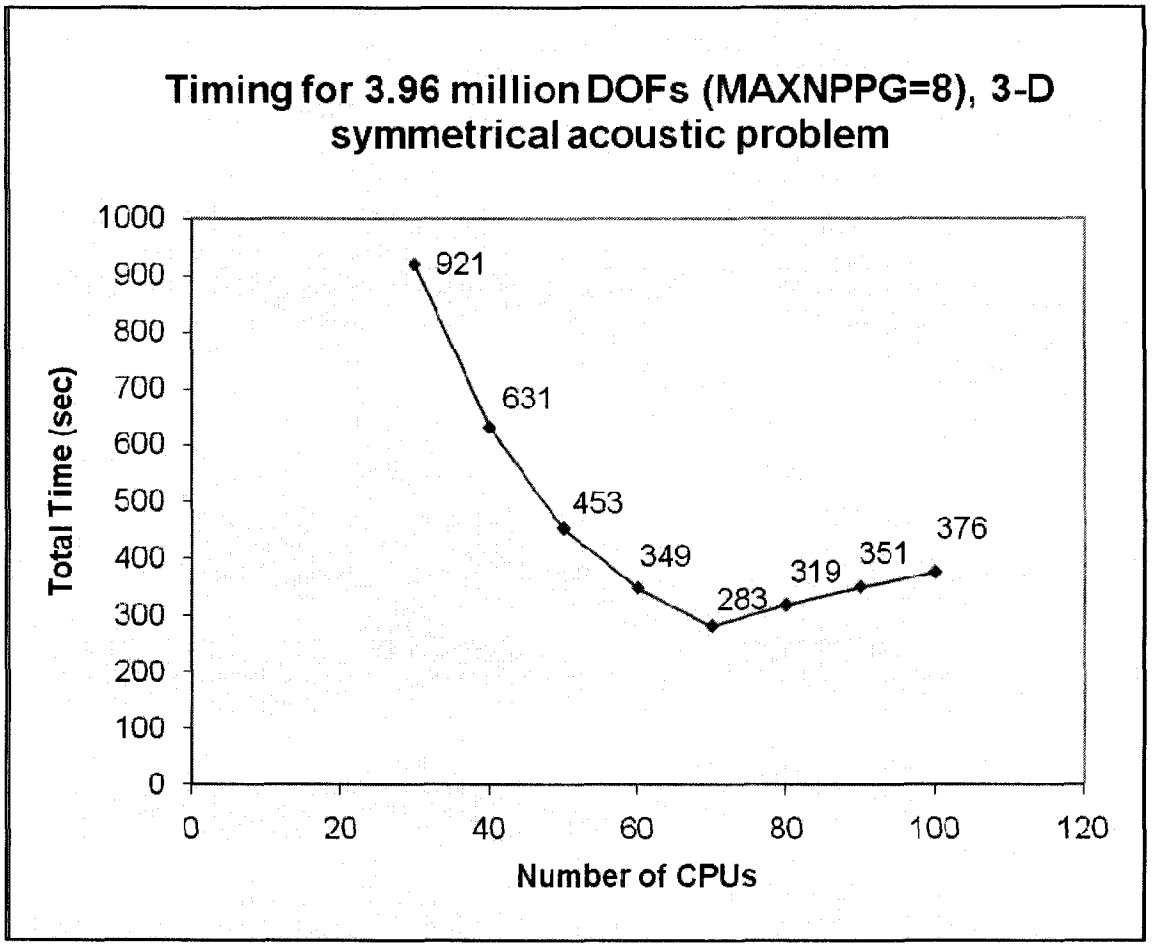


Figure 5.12: Timing for 3.96 million dofs (MAXNPPG=8), 3-D symmetrical acoustic example

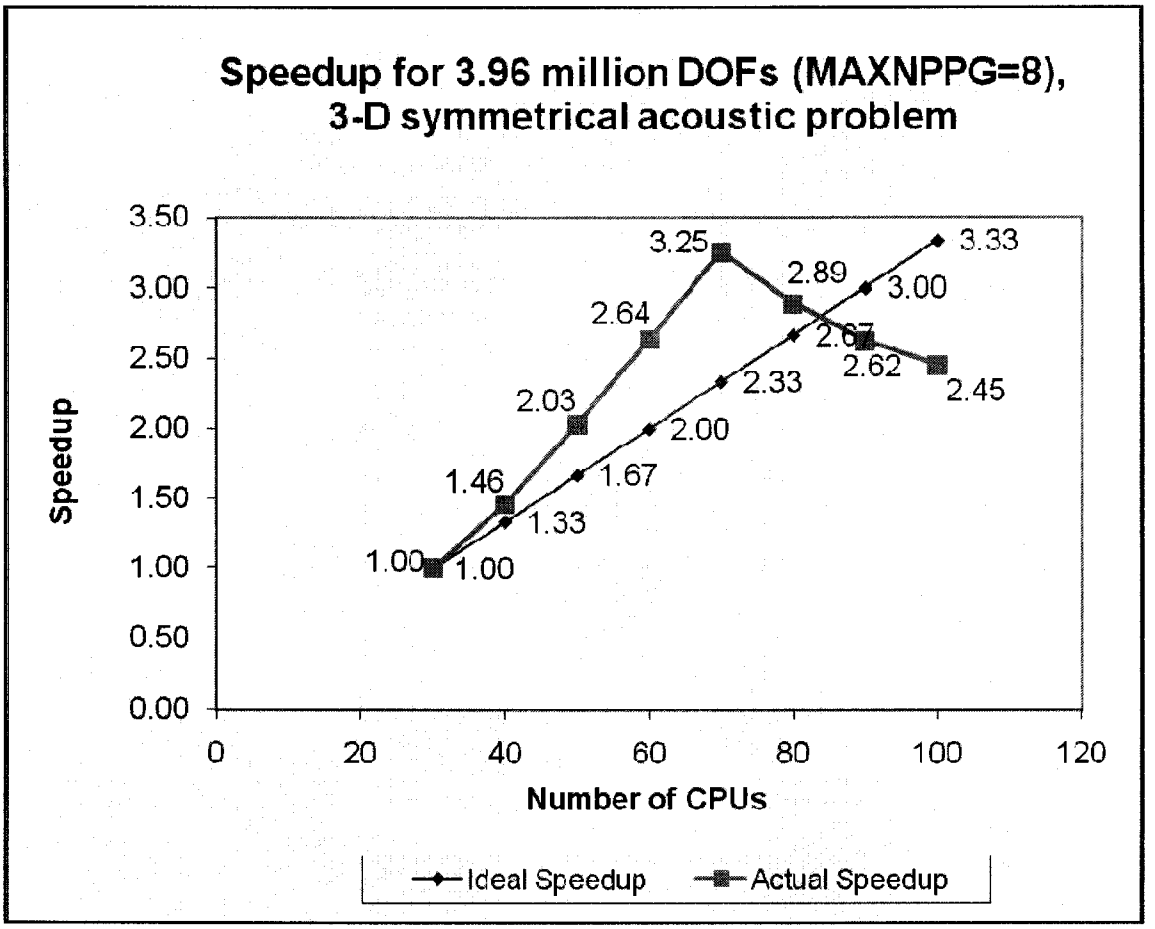


Figure 5.13: Speedup for 3.96 million dofs (MAXNPPG=8), 3-D symmetrical acoustic example

Table 5.7: Timing statistic for 3.96 million dofs (MAXNPPG=16), 3-D symmetrical acoustic example

3.96M (MAXNPPG=16)	10	20	30	40	50	60	70	80	90	100
TT	-	-	934	663	445	377	294	323	371	397
ISUP	-	-	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33
ASUP	-	-	1.00	1.41	2.10	2.48	3.18	2.89	2.52	2.35
MaxMem	-	-	1441	981	734	573	461	394	362	331
TOT_BDOF	-	-	104400	140400	176400	212400	248400	284400	320400	356400
MAX_BDOF	-	-	7200	7200	7200	7200	7200	7200	7200	7200
MAX_IDOF	-	-	129600	97200	79200	64800	54000	46800	43200	39600
PT	-	-	16.3	20.1	24.4	29	32.9	37.1	41.5	46
RT	-	-	3.1	2.2	1.8	1.4	1.1	0.9	0.9	0.8
AT	-	-	2.5	4.2	1.2	1.2	2.7	4.5	1.1	0.9
FT	-	-	618	268	154	96	57	38	30	24
BT	-	-	291	365	261	247	196	239	295	322
NIT	-	-	136	163	186	201	159	236	252	266
IT	-	-	1.4	1.2	0.6	0.4	0.6	0.3	0.2	0.4

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

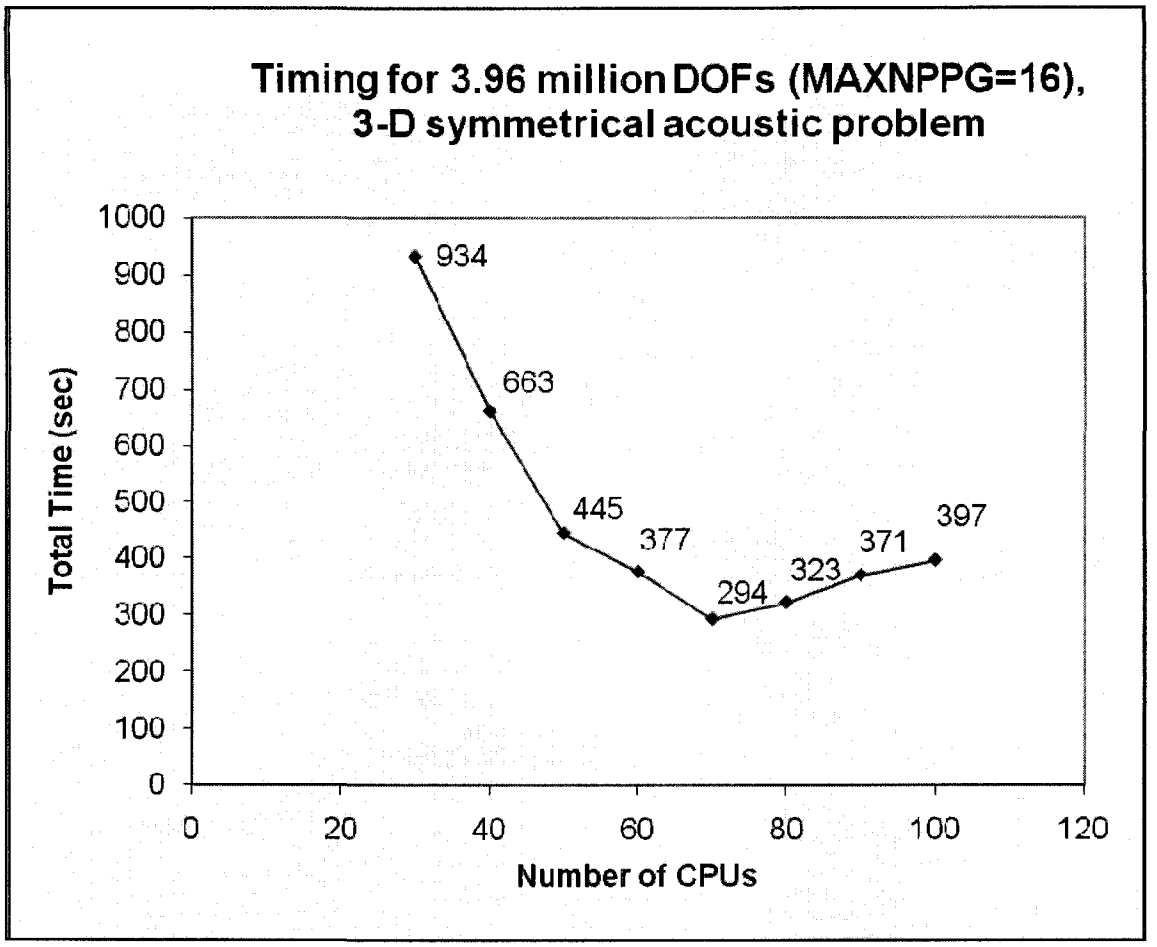


Figure 5.14: Timing for 3.96 million dofs (MAXNPPG=16), 3-D symmetrical acoustic example

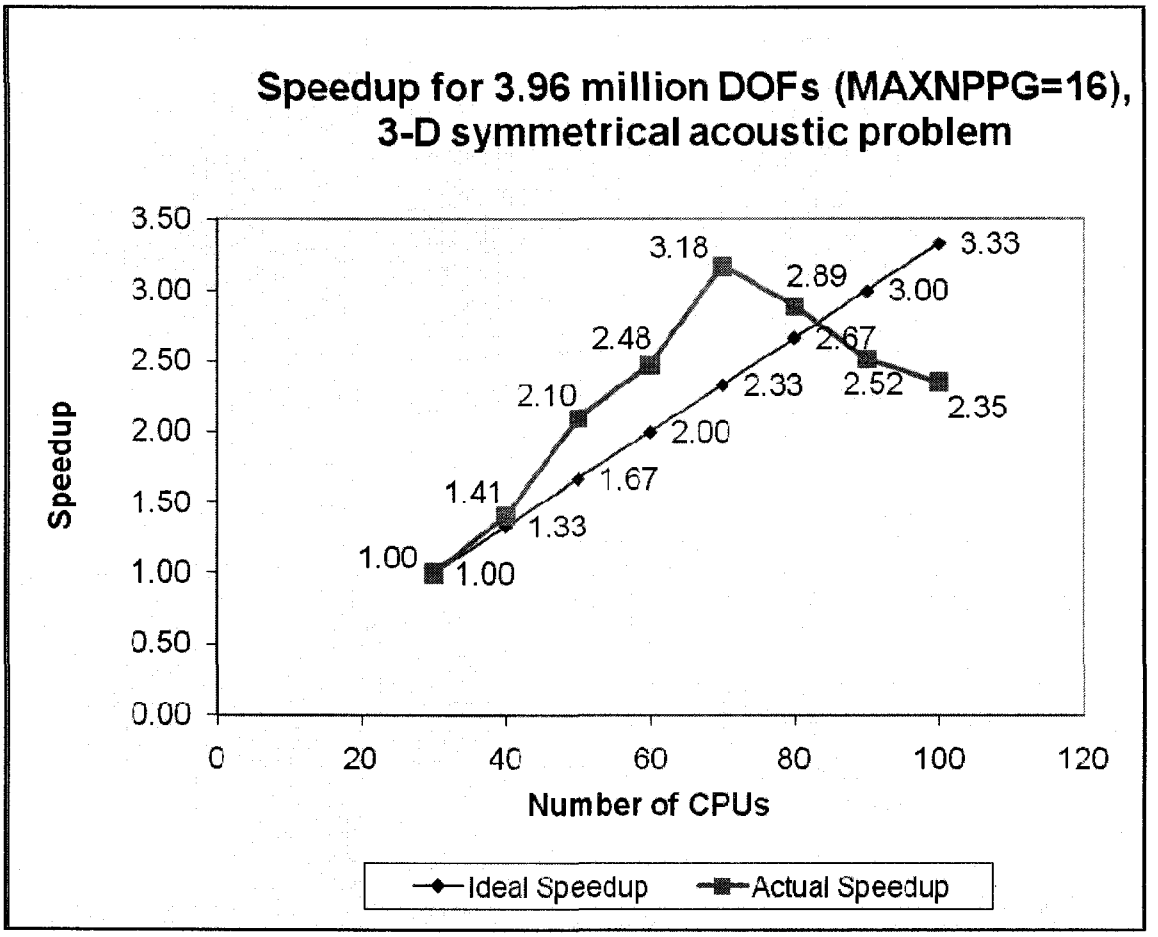


Figure 5.15: Speedup for 3.96 million dofs (MAXNPPG=16), 3-D symmetrical acoustic example

Table 5.8: Total time and memory used (in MB, shown within parentheses) of different MAXNPPG values for 3.96 million dofs, 3-D acoustic problem

3.96 M	10	20	30	40	50	60	70	80	90	100
MAXNPPG=1	-	-	899 sec (1453 MB)	604 (998)	420 (755)	324 (598)	250 (491)	259 (428)	279 (401)	274 (373)
MAXNPPG=8	-	-	921 (1442)	631 (981)	453 (735)	349 (575)	283 (463)	319 (396)	351 (365)	376 (334)
MAXNPPG=16	-	-	934 (1441)	663 (981)	445 (734)	377 (573)	294 (461)	323 (394)	371 (362)	397 (331)
MAXNPPG=32	-	-	807 (1441)	541 (981)	414 (733)	362 (572)	283 (461)	354 (393)	380 (361)	399 (330)

To solve large-scale problems, arrays in Parallel Preconditioned Conjugate Gradient subroutine are partitioned based on the discussion in chapter 4.4. Results of maximum number of processors per group, MAXNPPG, equals to 32, 1, 8 and 16 are represented in Table 5.3, 5.5, 5.6 and 5.7, respectively. Then, a combined version of total times and memory used for each MAXNPPG case are represented in Table 5.8. The advantage of this storage scheme is that the more the maximum processors in the group, the less the memory required during iterative solver. In addition, some computations can be computed in parallel among the processors in the group. However, the communication between the processors in the group will drastically increase as more processors are used. As a result, the total time in iterative solver will benefit this storage scheme at the beginning, and it will take more time the more processors are used in the process.

Table 5.9: Timing statistic for 3.96 million dofs (ParMETIS), 3-D symmetrical acoustic example

3.96M (ParMETIS)	10	20	30	40	50	60	70	80	90	100
TT	-	-	1230	716	647	635	607	529	493	576
ISUP	-	-	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33
ASUP	-	-	1.00	1.72	1.90	1.94	2.03	2.33	2.49	2.14
MaxMem	-	-	1517	1052	825	671	613	495	449	410
TOT_BDOF	-	-	111016	156938	199573	220964	242921	260995	276481	295721
MAX_BDOF	-	-	8581	9541	9749	8867	9624	8526	8090	8423
MAX_IDOF	-	-	133844	101971	81559	66166	57254	49873	44313	39558
PT	-	-	35.8	43.7	51.4	59.4	67.9	78	90.6	103
RT	-	-	3.2	2.3	1.6	1.3	1	1	0.8	0.7
AT	-	-	6	15.5	15.2	13.6	6.3	2.6	4.2	4.9
FT	-	-	634	290	155	143	82	75	47	47
BT	-	-	547	362	421	415	447	368	348	418
NIT	-	-	254	226	252	313	313	307	270	335
IT	-	-	1.3	1.2	0.5	0.5	0.5	0.5	0.4	0.4

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

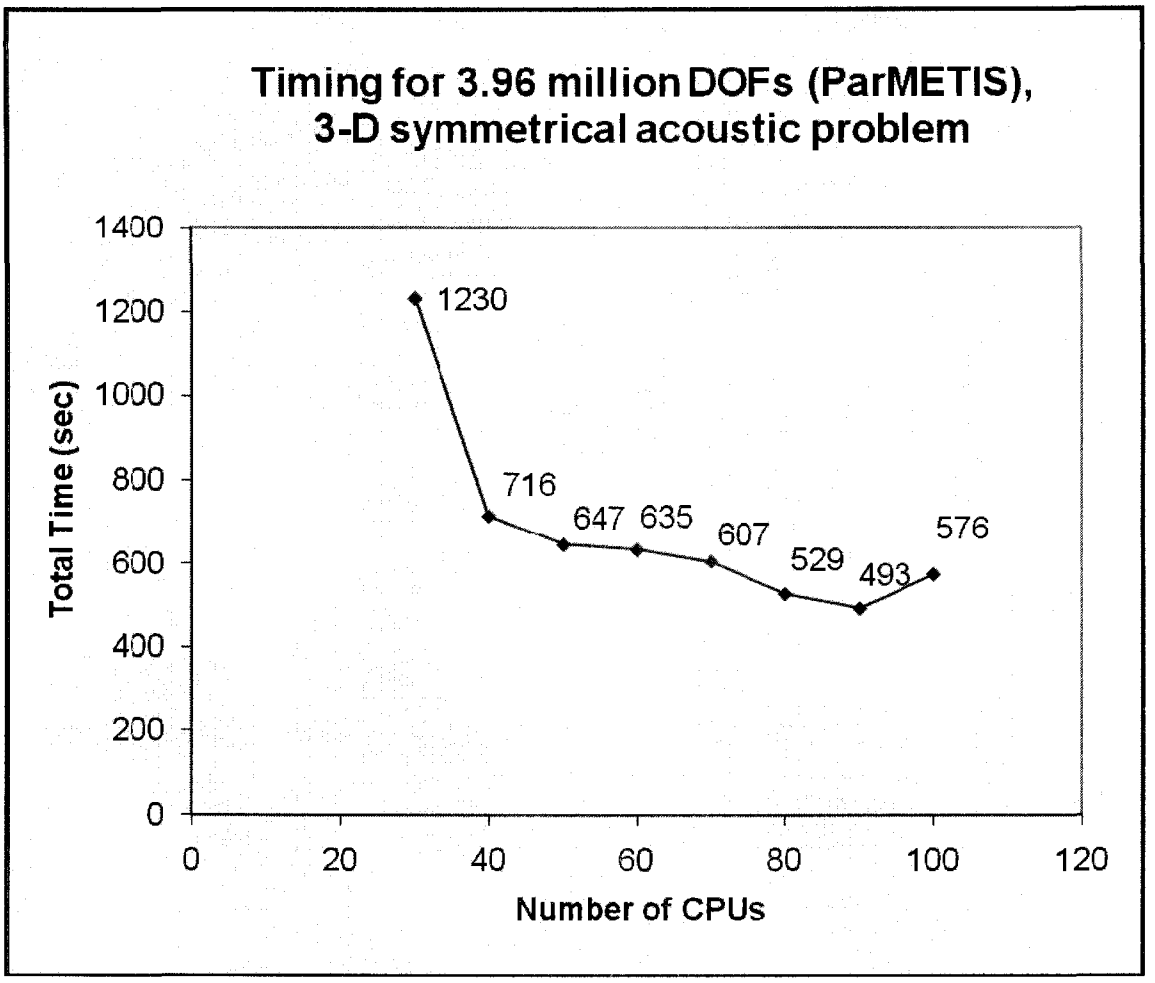


Figure 5.16: Timing for 3.96 million dofs (ParMETIS), 3-D symmetrical acoustic example

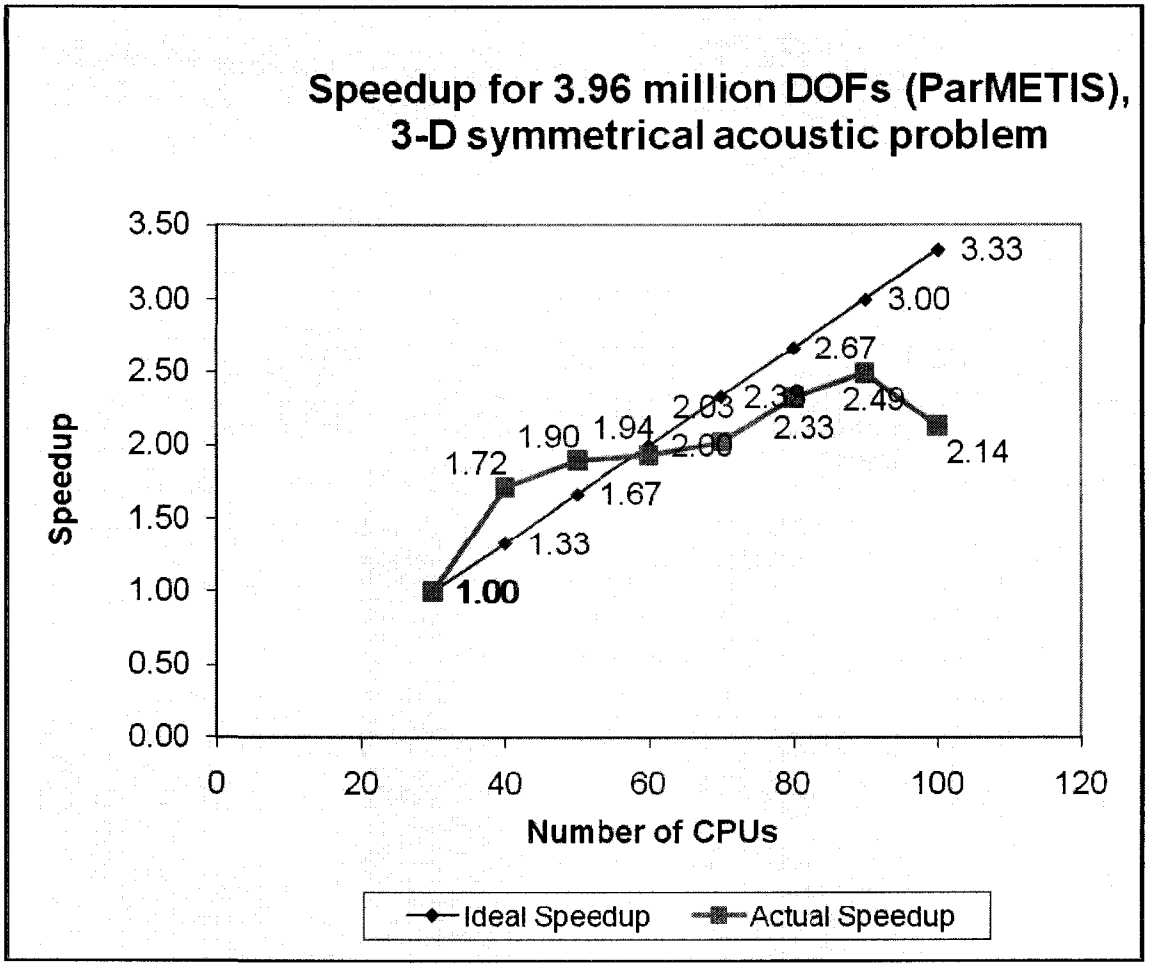


Figure 5.17: Speedup for 3.96 million dofs (ParMETIS), 3-D symmetrical acoustic example

Table 5.10: Timing statistic for 3.96 million dofs (using unsymmetrical solver), 3-D symmetrical acoustic example

3.96M (Unsym)	10	20	30	40	50	60	70	80	90	100
TT	-	-	-	1472	1114	929	796	833	847	884
ISUP	-	-	-	1.00	1.25	1.50	1.75	2.00	2.25	2.50
ASUP	-	-	-	1.00	1.32	1.58	1.85	1.77	1.74	1.67
MaxMem	-	-	-	1715	1285	1009	821	713	670	626
TOT_BDOF	-	-	-	140400	176400	212400	248400	284400	320400	356400
MAX_BDOF	-	-	-	7200	7200	7200	7200	7200	7200	7200
MAX_IDOF	-	-	-	97200	79200	64800	54000	46800	43200	39600
PT	-	-	-	20	24.4	28.6	32.8	37.3	41.5	46.1
RT	-	-	-	2.2	18	1.4	1.1	0.9	0.9	0.8
AT	-	-	-	4.4	1.2	1.3	2.7	4.7	1.1	0.9
FT	-	-	-	546	321	191	96	71	51	40
BT	-	-	-	895	763	704	660	716	750	794
NIT	-	-	-	17	19	20	20	21	23	24
IT	-	-	-	1	0.8	0.6	0.5	0.4	0.3	0.2

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

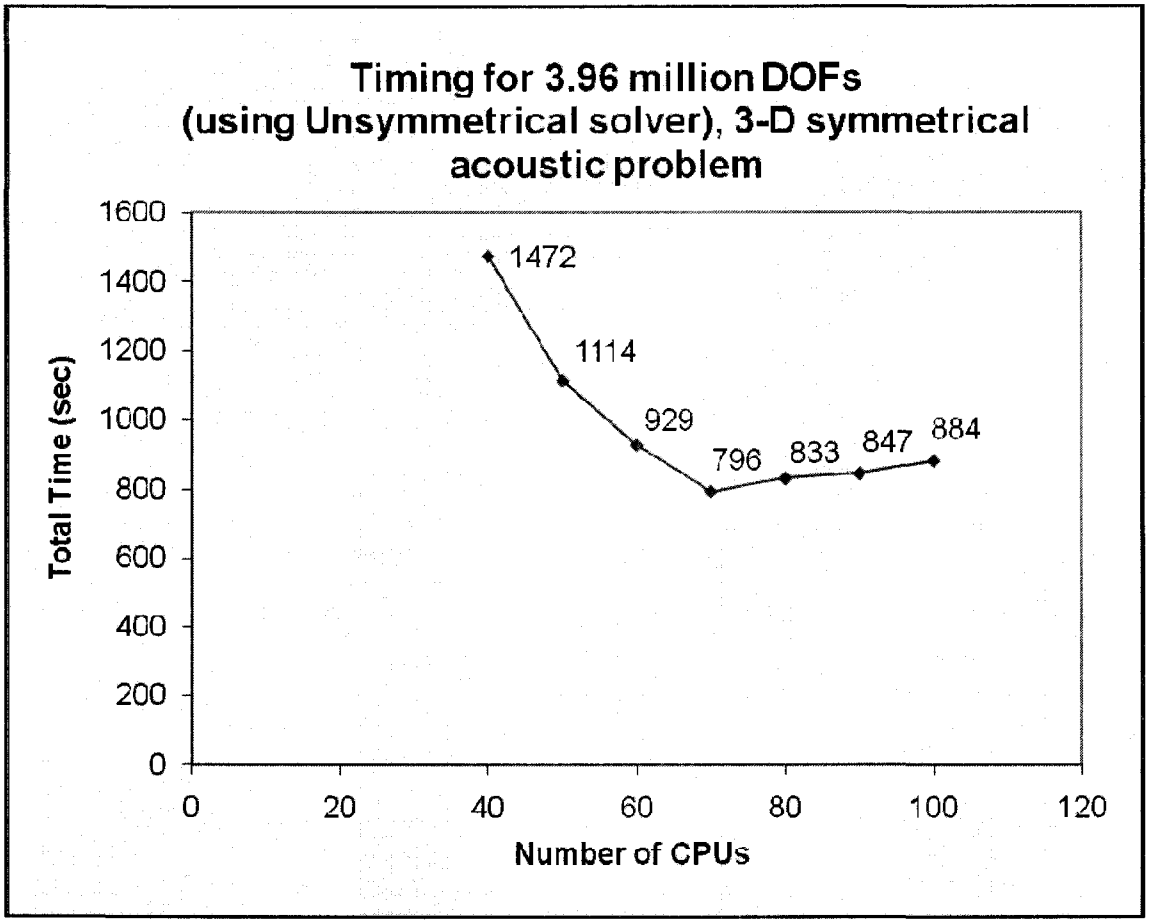


Figure 5.18: Timing for 3.96 million dofs (using unsymmetrical solver), 3-D symmetrical acoustic example

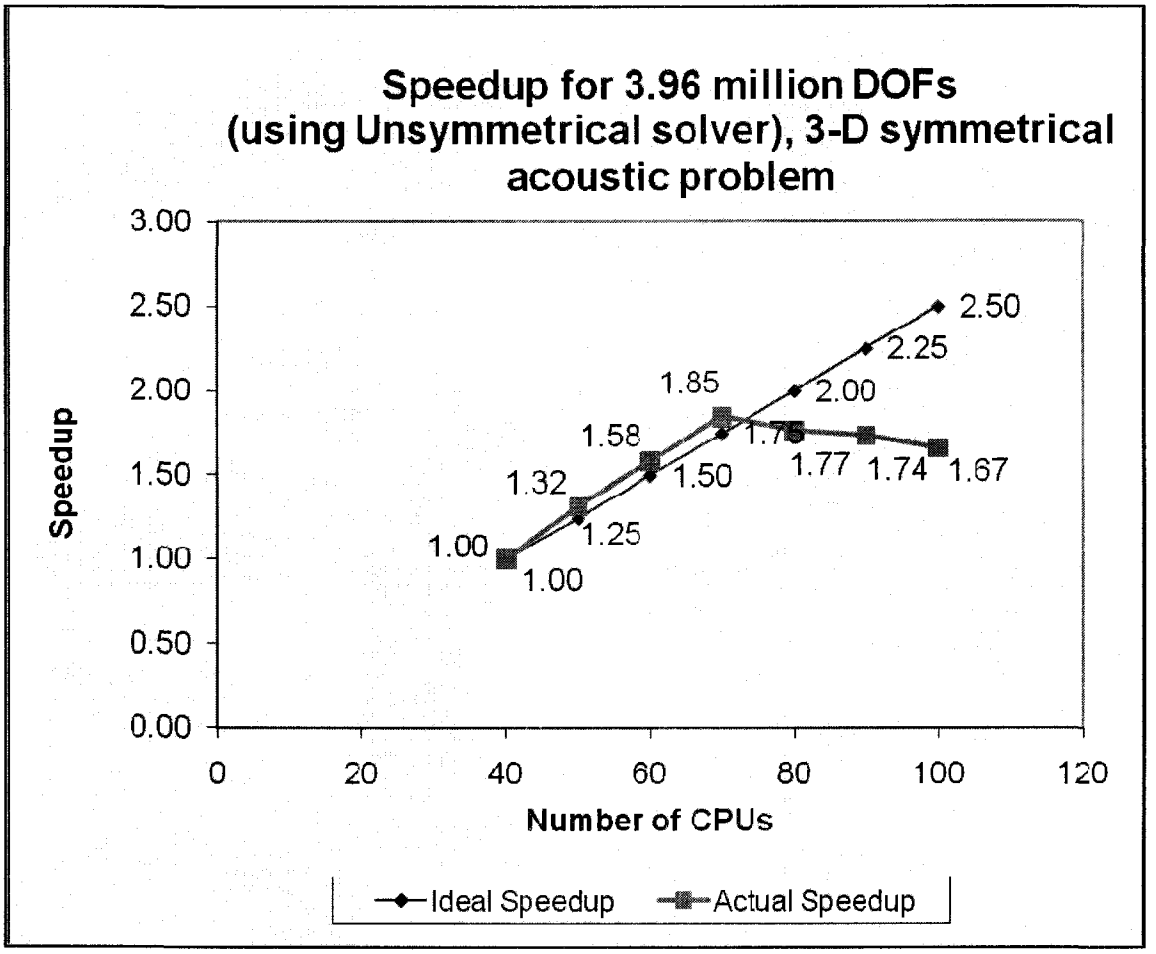


Figure 5.19: Speedup for 3.96 million dofs (using unsymmetrical solver), 3-D symmetrical acoustic example

Table 5.11: Total time and memory used (in MB, shown within parentheses) of different partitioning schemes for 3.96 million dofs, 3-D acoustic problem

3.96 M	10	20	30	40	50	60	70	80	90	100
Siroj's Scheme	-	-	899 sec (1453 MB)	604 (998)	420 (755)	324 (598)	250 (491)	259 (428)	279 (401)	274 (373)
ParMETIS	-	-	1230 (1517)	716 (1052)	647 (825)	635 (671)	607 (613)	529 (495)	493 (449)	576 (410)

As mentioned earlier, there are 2 partitioning schemes used in this work. A total time comparison of these two schemes is represented in Table 5.11. From the results, the author's scheme to break the domain is better than ParMETIS in terms of both total time and memory requirements. This is because the example demonstated here is a simple shape structure, and the subdomains partitioned from the author's scheme are well-balanced. As a result, the idle time due to unbalanced workload is less than the subdomained partitioned from ParMETIS algorithm. ParMETIS partitioning scheme provided in this work is for use with irregular-shaped structures.

Table 5.12: Total time and memory used (in MB, shown within parentheses) of different iterative solvers for 3.96 million dofs, 3-D acoustic problem

3.96 M	10	20	30	40	50	60	70	80	90	100
Parallel PCG	-	-	899 sec (1453 MB)	604 (998)	420 (755)	324 (598)	250 (491)	259 (428)	279 (401)	274 (373)
Paralle FGMRES(m)	-	-	-	1472 (1715)	1114 (1285)	929 (1009)	796 (821)	833 (713)	847 (670)	884 (626)

In Table 5.12, a 3.96 million degrees of freedom problem size is solved by both parallel Preconditioned Conjugate Gradient (PCG) and parallel Flexible Generalized Minimum Residual (FGMRES(m)). Total times using parallel FGMRES(m) are about 2-3.5 times slower than using parallel PCG. In addition, unsymmetrical factorization time is

about 2 times slower than symmetrical factorization time, and FGMRES(m) is about 3 to 3.5 times slower than PCG. In terms of memory, parallel FGMRES(m) requires almost two times more memory than parallel PCG, since the lower triangular part of the subdomain's coefficient matrices is used by FGMRES(m).

5.2 Example 2 – Two dimensional acoustic finite element model with flow

In this example, a 2-D acoustic panel example with flow shown in Figure 5.20 is demonstrated. The example is modeled with rectangular elements, and the boundary edge is along the X-axis line. Each node has 4 degrees of freedom. The size of the problem is determined by the product of MA and NA (i.e. $N = 4 \cdot MA \cdot NA$). Results are presented for 4 grids (N = 1.0, 3.2, 6.0 and 8.4 million degrees of freedom). The results are obtained from the ODU Wilbur cluster, which has 64 nodes, and each node has 2 processors and 4GB of memory.

There are several important remarks from the results presented in Table 5.13 to 5.16 and Figure 5.21 to 5.28. Please note that some remarks below are repeated from the previous example.

Most computational time occurs in the factorization phase and the boundary displacements solving phase.

Partitioning time tends to increase when more processors are used. Although most of the steps explained in chapter 6.3 are running independently on each processor, the first step is running sequentially and has data communication among processors. Therefore, increasing the number of processors adds more communication time to the total time.

Reordering, assembly, factorization and interior degrees of freedom solving times

decrease when the number of processors increases. The reason is that the size of subdomains will be smaller when the domain is partitioned into more subdomains.

The maximum memory used by the program is linearly reduced as more processors are used.

Timing in the boundary degrees of freedom solving phase depends on the number of iterations used during the phase. In contrast, an increasing number of iterations used during the boundary degrees of freedom solving phase is not necessary influenced by an increase of processors used.

Although the factorization time is significantly reduced when using more processors, the number of boundary degrees of freedom will also increase. This is because increasing the number of subdomains will introduce more edge cuts to the domain. As a result, boundary degrees of freedom solving time using the iterative solver will also increase.

From the factorization time of 1 million DOFs example using both symmetrical and unsymmetrical factorization algorithm, although the size of the original matrices are about the same, factorization time using the symmetrical solver is at most 7 times slower than using the unsymmetrical solver. This is mainly because the amount of fill-in terms that occurred in the factorization phase of the 3-D example is much more than in the 2-D example.

The numbers of iterations represented in Table 5.13 to 5.16 refer to the outer iteration. Basically, each outer iteration includes up to 30 inner iterations, depending on the error tolerance set in the inner iterative solver (see chapter 4.5 for details).

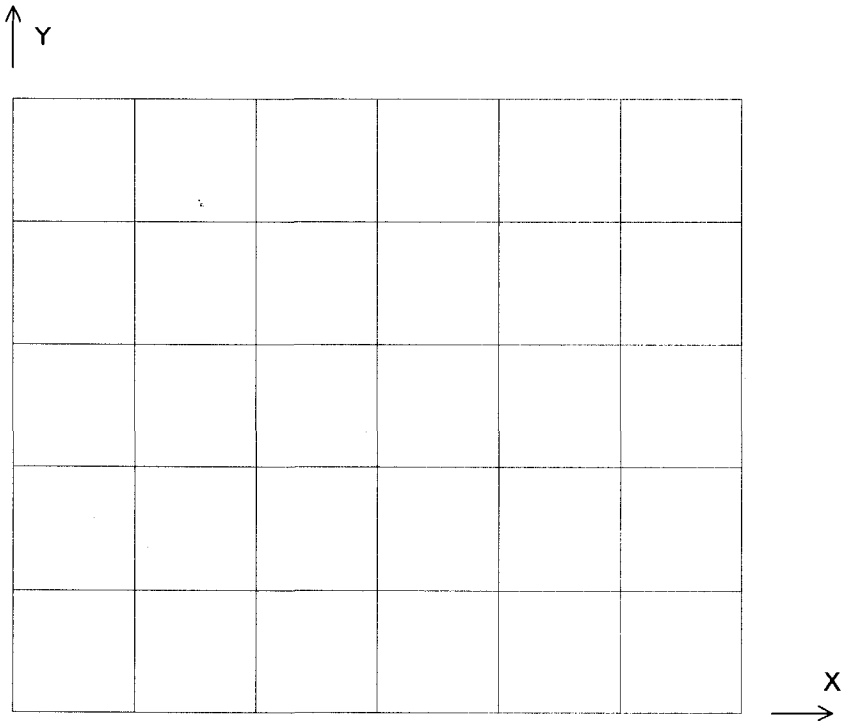


Figure 5.20: 2-D unsymmetrical acoustic example

Table 5.13: Timing statistic for 1 million dofs, 2-D unsymmetrical acoustic example

1M	10	20	30	40	50	60	70	80	90	100
TT	363	204	145	160	152	120	149	150	143	159
ISUP	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
ASUP	1.00	1.78	2.50	2.27	2.39	3.03	2.44	2.42	2.54	2.28
MaxMem	608	290	189	146	119	101	90	80	74	70
TOT_BDOF	7984	12964	16928	20240	23544	25832	28308	30772	32728	34696
MAX_BDOF	1804	1404	1464	1280	1136	1040	968	904	848	800
MAX_IDOF	100000	50000	33264	24928	20336	16632	14152	12348	11000	10000
PT	0.3	3.6	1	1.3	1.7	2	2.5	2.9	3.2	3.7
RT	0.4	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
AT	0.5	0.3	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1
FT	39	17	9	5	3	3	2	2	2	1
BT	323	182	134	153	147	114	143	144	137	153
NIT	21	25	28	36	40	38	43	45	43	47
IT	0.4	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

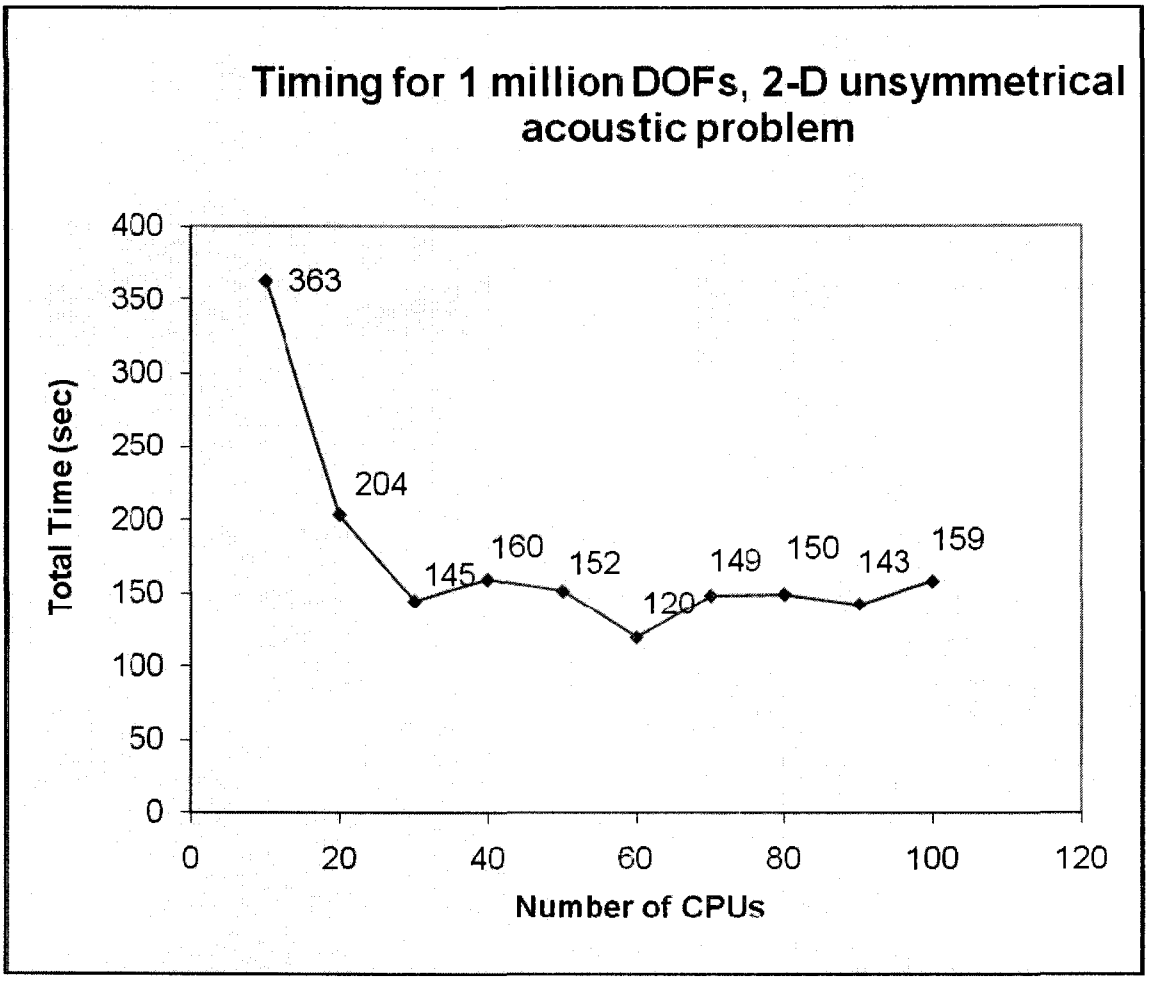


Figure 5.21: Timing for 1 million dofs, 2-D unsymmetrical acoustic example

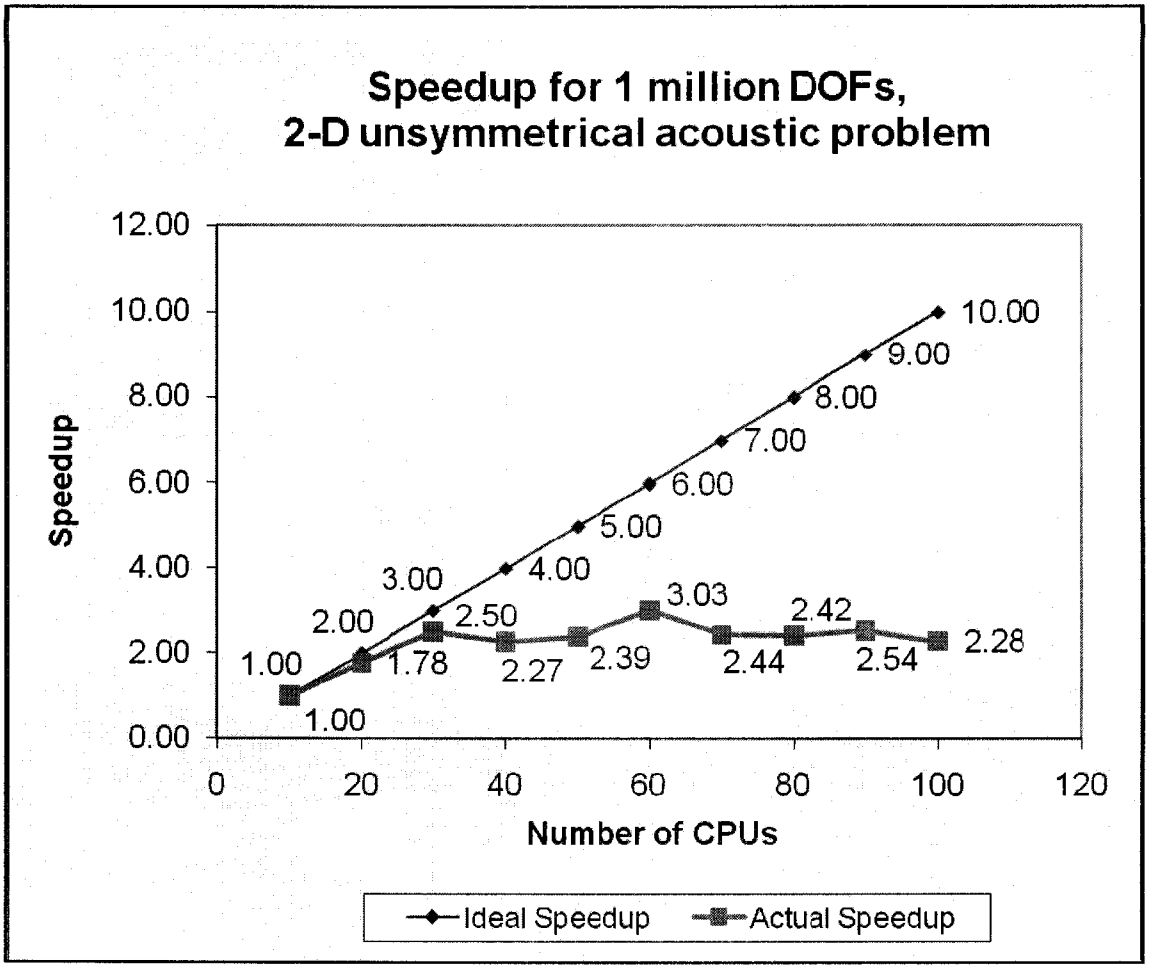


Figure 5.22: Speedup for 1 million dofs, 2-D unsymmetrical acoustic example

Table 5.14: Timing statistic for 3.2 million dofs, 2-D unsymmetrical acoustic example

3.2M	10	20	30	40	50	60	70	80	90	100
TT	-	870	730	610	551	498	488	456	510	485
ISUP	-	1.00	1.50	2.00	2.50	3.00	3.50	4.00	4.50	5.00
ASUP	-	1.00	1.19	1.43	1.58	1.75	1.78	1.91	1.71	1.79
MaxMem	-	1066	682	515	430	348	311	272	247	227
TOT_BDOF	-	22364	30344	35640	40944	46248	50204	54172	58144	62112
MAX_BDOF	-	2404	2140	2296	2008	1864	1744	1600	1528	1440
MAX_IDOF	-	160000	106400	80784	66000	53600	46332	40000	35640	32000
PT	-	1.6	2.5	3.3	4.2	5	6	7	7.9	8.9
RT	-	0.7	0.4	0.3	0.2	0.2	0.2	0.1	0.1	0.2
AT	-	1	0.7	0.5	0.5	0.3	0.3	0.3	0.3	0.2
FT	-	110	50	27	20	15	13	10	9	7
BT	-	755	675	577	524	475	468	436	491	466
NIT	-	29	37	42	44	47	48	49	53	55
IT	-	0.8	0.4	0.3	0.3	0.2	0.2	0.2	0.2	0.1

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

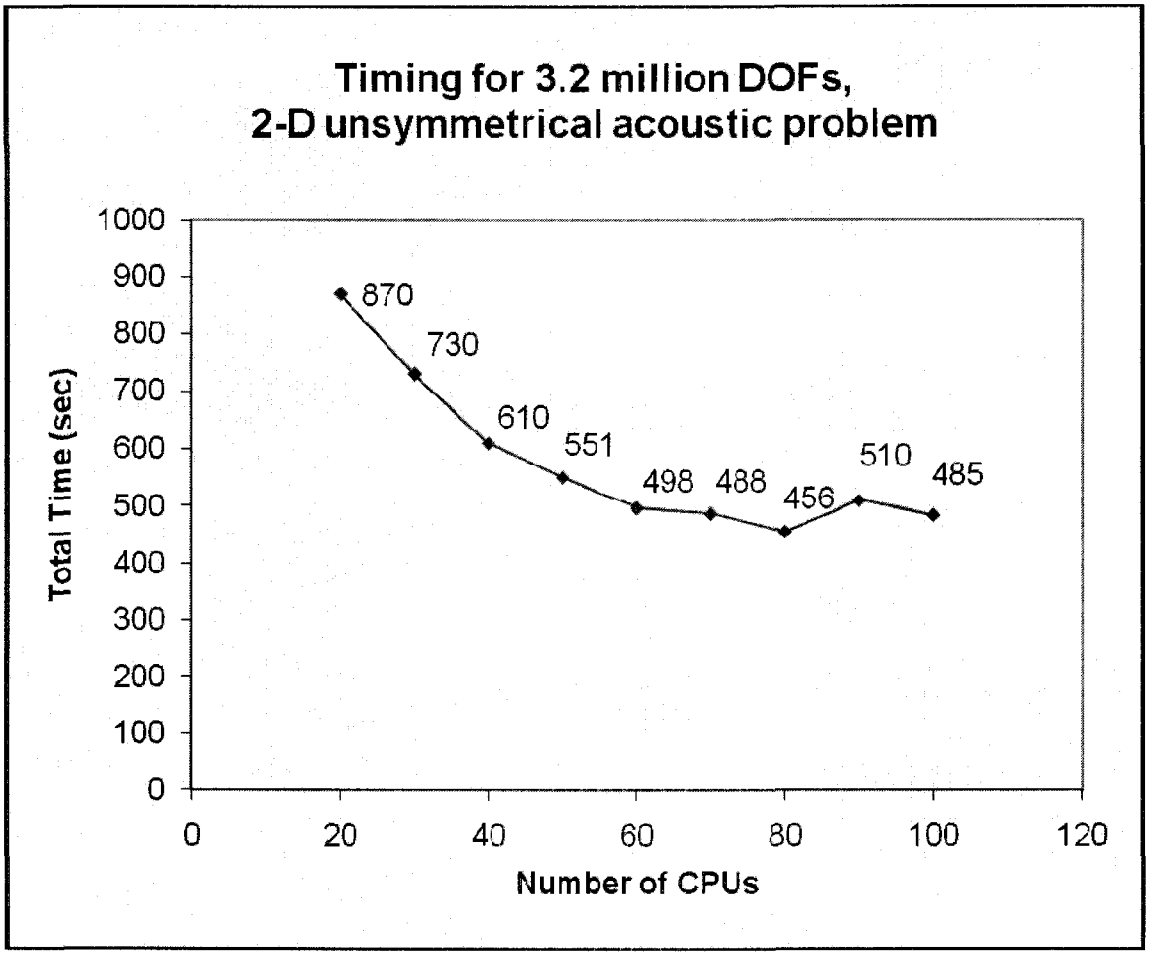


Figure 5.23: Timing for 3.2 million dofs, 2-D unsymmetrical acoustic example

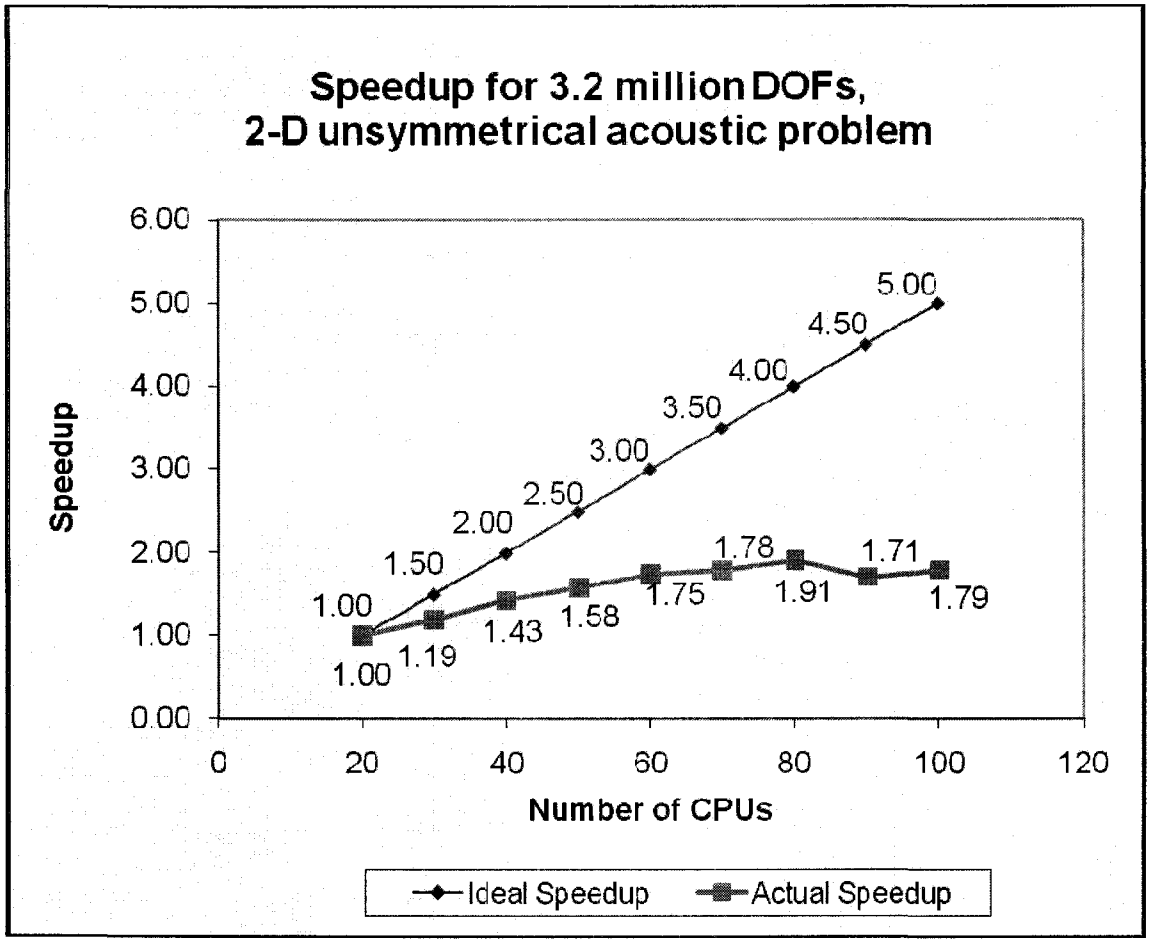


Figure 5.24: Speedup for 3.2 million dofs, 2-D unsymmetrical acoustic example

Table 5.15: Timing statistic for 6 million dofs, 2-D unsymmetrical acoustic example

6M	10	20	30	40	50	60	70	80	90	100
TT	-	-	1561	1578	1375	1081	1122	1020	1066	1038
ISUP	-	-	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33
ASUP	-	-	1.00	0.99	1.14	1.44	1.39	1.53	1.46	1.50
MaxMem	-	-	1383	1064	851	680	618	530	478	443
TOT_BDOF	-	-	41528	49504	57476	63432	69404	75372	80528	85296
MAX_BDOF	-	-	3600	3144	2776	2536	2376	2200	2072	1960
MAX_IDOF	-	-	200000	152832	124176	99600	87612	75000	66240	60000
PT	-	-	4.3	5.8	7.4	8.7	10.3	11.9	16.5	15.3
RT	-	-	0.9	0.7	0.5	0.4	0.3	0.3	0.3	0.3
AT	-	-	1.2	1	0.8	0.6	0.5	0.5	0.5	0.4
FT	-	-	142	80	58	40	33	30	29	20
BT	-	-	1409	1487	1305	1028	1076	975	1016	1000
NIT	-	-	40	53	57	54	61	61	64	68
IT	-	-	0.9	0.7	0.6	0.5	0.4	0.4	0.3	0.3

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

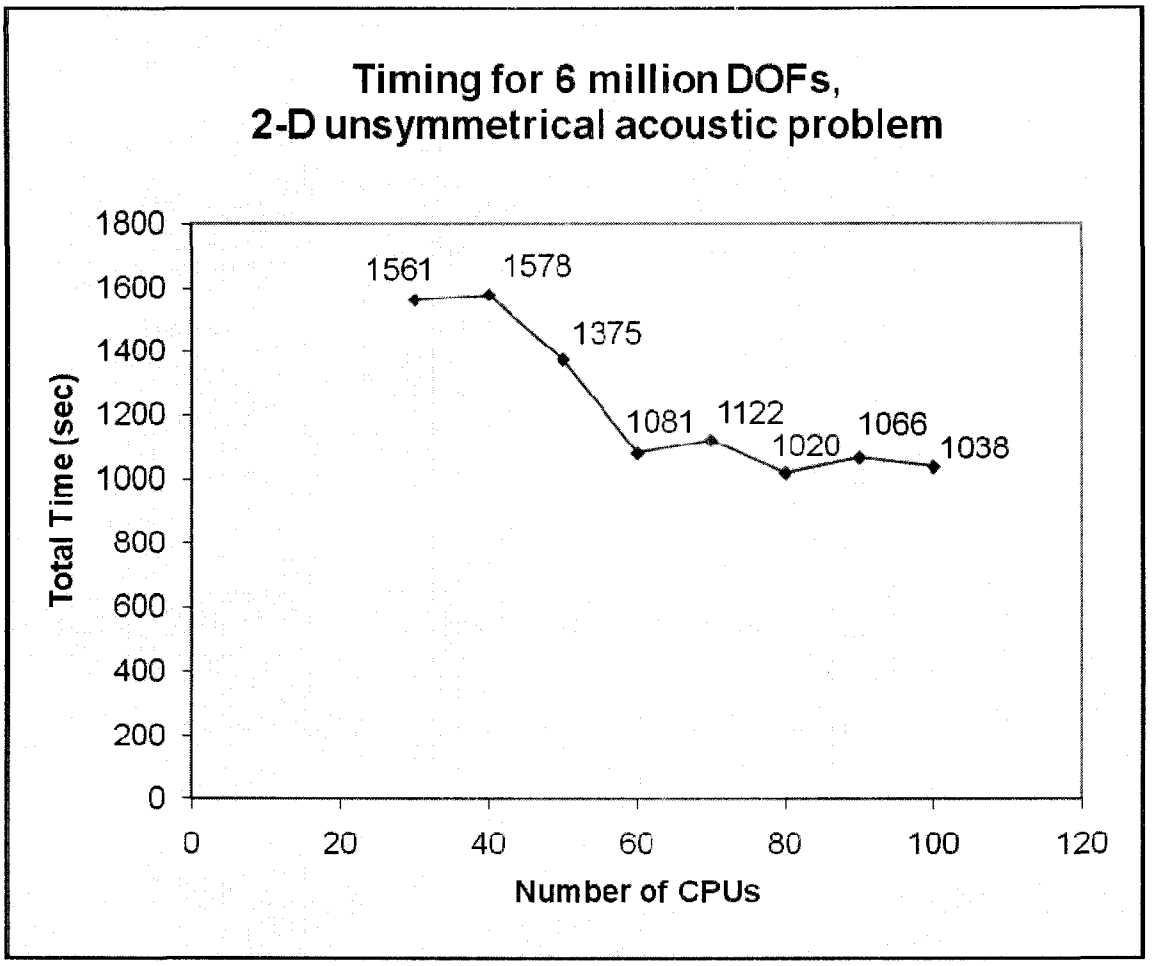


Figure 5.25: Timing for 6 million dofs, 2-D unsymmetrical acoustic example

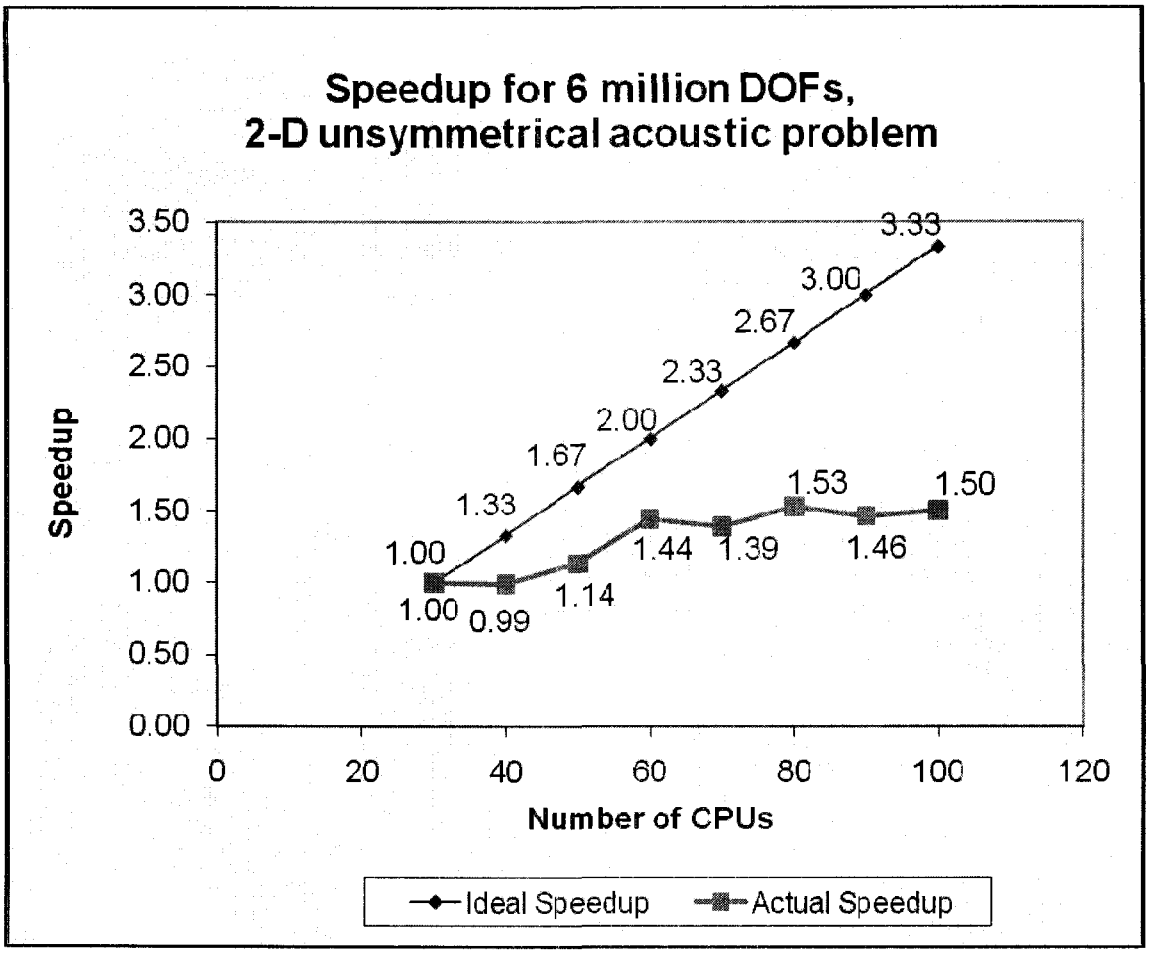


Figure 5.26: Speedup for 6 million dofs, 2-D unsymmetrical acoustic example

Table 5.16: Timing statistic for 8.4 million dofs, 2-D unsymmetrical acoustic example

8.4M	10	20	30	40	50	60	70	80	90	100
TT	-	-	-	2372	2044	1545	1636	1508	1500	1468
ISUP	-	-	-	1.00	1.25	1.50	1.75	2.00	2.25	2.50
ASUP	-	-	-	1.00	1.16	1.54	1.45	1.57	1.58	1.62
MaxMem	-	-	-	1562	1235	985	870	761	683	625
TOT_BDOF	-	-	-	58440	67744	75032	82004	88972	95328	100896
MAX_BDOF	-	-	-	3712	3280	3000	2816	2600	2456	2320
MAX_IDOF	-	-	-	213440	173536	140000	122496	105000	92960	84000
PT	-	-	-	8	9.9	11.9	16.9	19.1	18.5	20.3
RT	-	-	-	0.9	0.7	0.6	0.5	0.5	0.4	0.4
AT	-	-	-	1.6	1.3	0.8	0.8	0.7	0.6	0.5
FT	-	-	-	133	85	85	52	61	37	32
BT	-	-	-	2225	1942	1444	1562	1424	1440	1410
NIT	-	-	-	56	60	57	64	66	67	72
IT	-	-	-	1.1	0.8	0.6	0.6	0.5	0.5	0.3

TT	Total Time (sec)
ISUP	Ideal Speed up
ASUP	Actual Speed Up
MaxMem	Maximum memory used on a running node (MB)
TOT_BDOF	Total system boundary degrees of freedom
MAX_BDOF	Maximum Boundary degrees of freedom on a running node
MAX_IDOF	Maximum Interior degrees of freedom on a running node
PT	Partitioning domain into subdomains time
RT	Reordering time
AT	Assembly time
FT	Factorization of Kii time
BT	Boundary degrees of freedom solving time
NIT	Number of iterations
IT	Interior degrees of freedom solving time

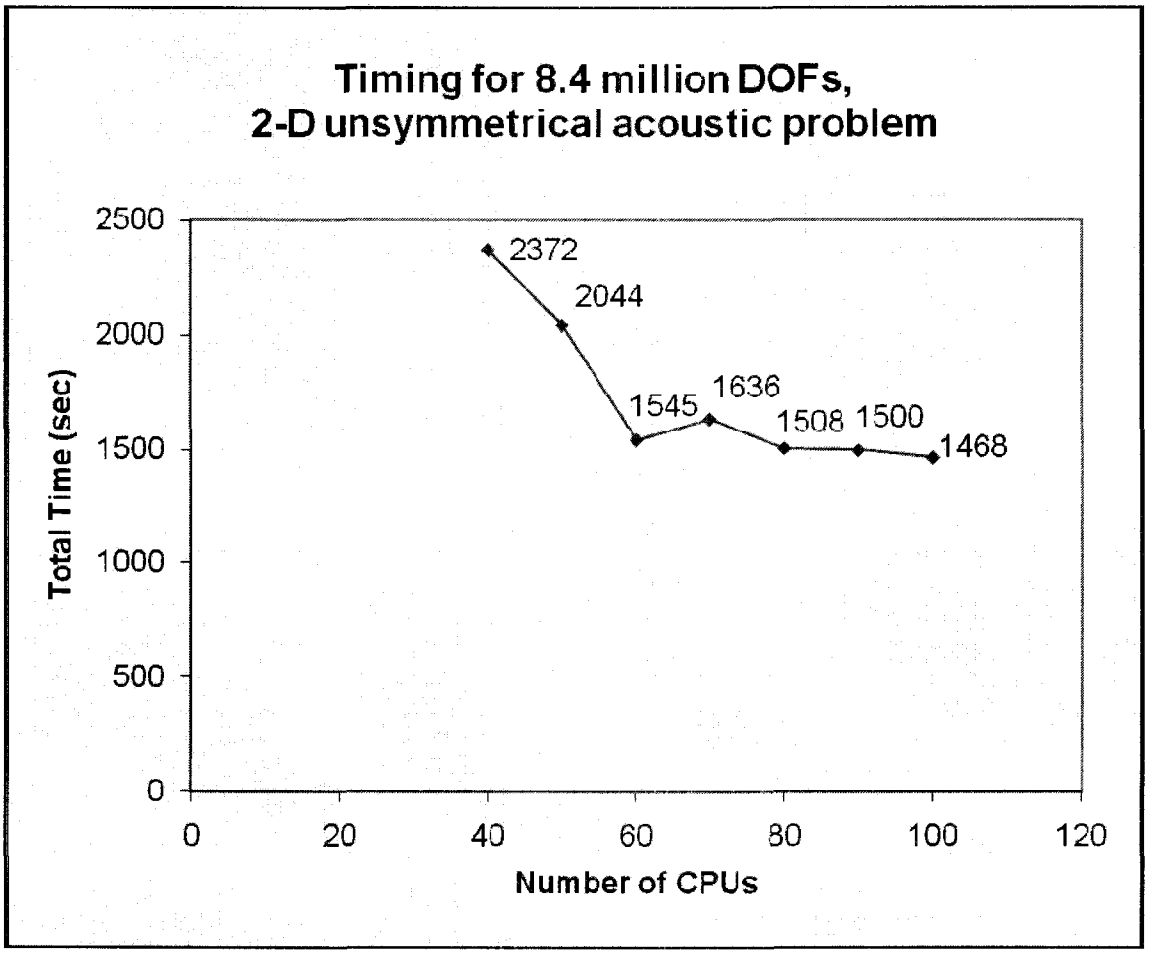


Figure 5.27: Timing for 8.4 million dofs, 2-D unsymmetrical acoustic example

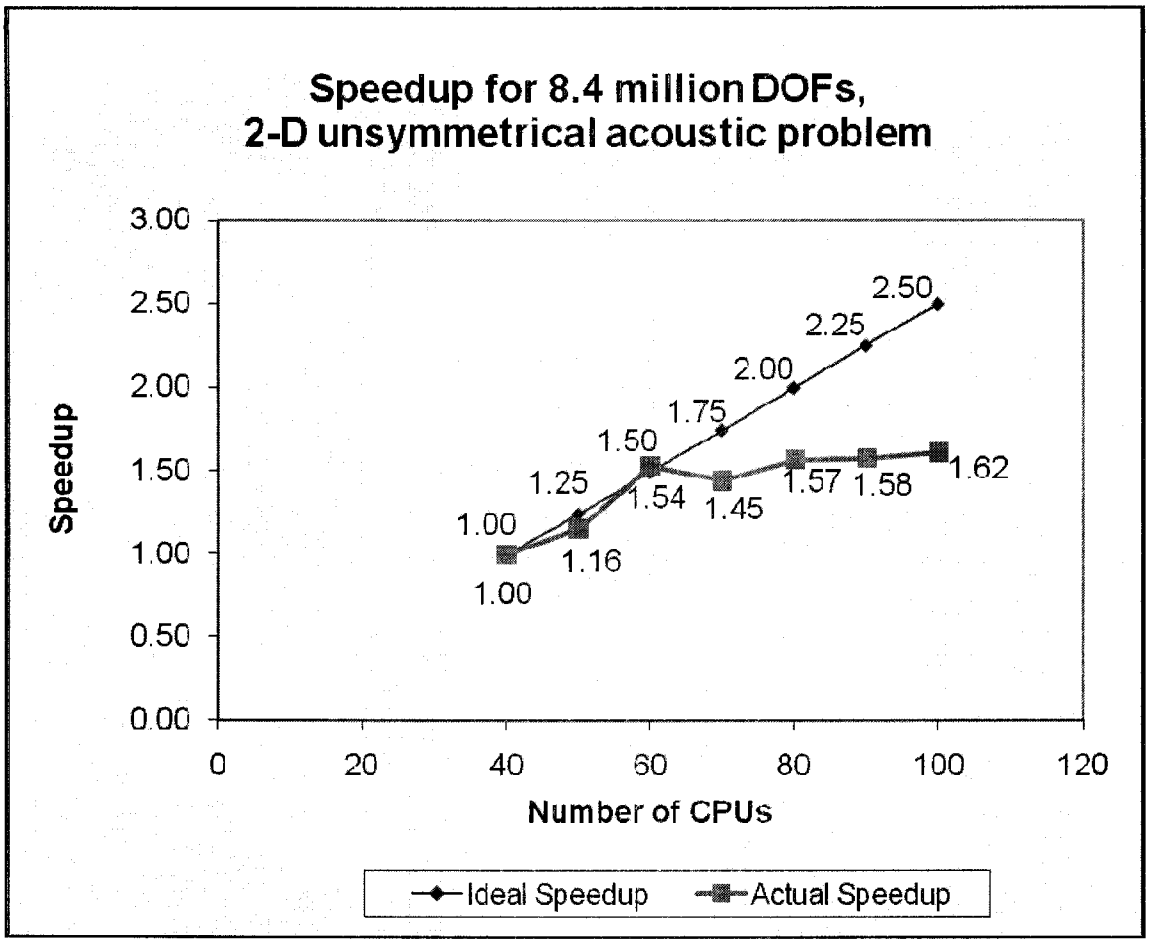


Figure 5.28: Speedup for 8.4 million dofs, 2-D unsymmetrical acoustic example

5.3 Example 3 – Three dimensional symmetrical acoustic example with 40 MPC equations

The previously discussed example 1 (3-D symmetrical acoustic with 2.5 million dofs) is reconsidered here. In this example, however, 40 MPC equations are included (see Appendix D) for more details of these 40 MPC equations. Numerical performance of the developed parallel-sparse FE-DD solver is summarized in Table 5.17, which clearly shows a dramatic reduction in both computational time and computer memory requirements as the number of processors are increased.

Table 5.17: Timing statistic for 2.5 million dofs, 3-D symmetrical acoustic with 40 MPC equations example

2.5M (MPC)	20	30	40	50
Time(sec)	761	402	253	218
Ideal Speedup	1.00	1.50	2.00	2.50
Actual Spedup	1.00	1.89	3.01	3.49
Max memory (MB)	1409	799	560	416

CHAPTER VI

DETAILED STEPS IN MPI/FORTRAN DD FORMULATION

6.1 Data partitioning for user input

For large-scale applications on distributed memory machines, the available memory might not be enough if the input data (i.e. element connectivity, joint coordinates, material properties set of elements, etc.) are stored on only one processor. For instance, the memory required to store only the node coordinates of a 50 million degrees of freedom, 3-D acoustic problem is about 1.2 Gigabytes, which is more than half of the memory available for each processor on Wilber cluster. Therefore, a special storage scheme is used to store large input data. In other words, the input data, which are element connectivity, node coordinates, and material properties set of elements information, are partitioned and stored among the processors before calling Domain Decomposition Finite Element Analysis subroutine.

To illustrate the storage scheme used in this work, a 2-dimensional, 16 nodes, 9 elements example is provided in Figure 6.1. Then, element connectivity, node coordinates and material set of the elements are expressed in Tables 6.1 and 6.2.

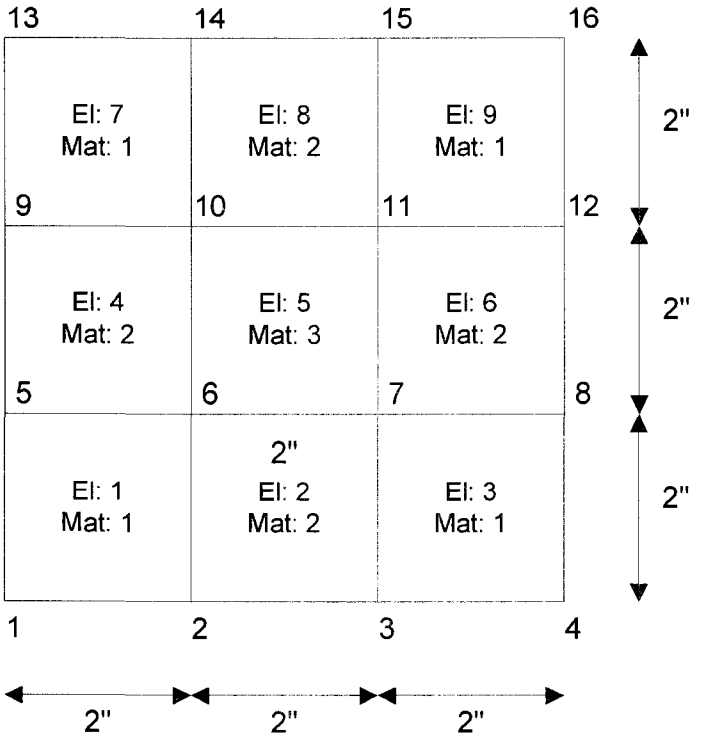


Figure 6.1: Simple 16 nodes, 9 elements example

Table 6.1: Element connectivity and material set of elements of the example in Figure 6.1

Element	1st node	2nd node	3rd node	4th node	Material set
1	1	5	6	2	1
2	2	6	7	3	2
3	3	7	8	4	1
4	5	9	10	6	2
5	6	10	11	7	3
6	7	11	12	8	2
7	9	13	14	10	1
8	10	14	15	11	2
9	11	15	16	12	1

Table 6.2: Node coordinates of the example in Figure 6.1

Node	X coordinate	Y coordinate
1	0.000	0.000
2	2.000	0.000
3	4.000	0.000
4	6.000	0.000
5	0.000	2.000
6	2.000	2.000
7	4.000	2.000
8	6.000	2.000
9	0.000	4.000
10	2.000	4.000
11	4.000	4.000
12	6.000	4.000
13	0.000	6.000
14	2.000	6.000
15	4.000	6.000
16	6.000	6.000

Table 6.1 can, in fact, be described as in Figure 6.2, where columns represent node

number and rows represent element number. The X symbol denotes the association of nodes in the element.

Node \ Element	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	X	X			X	X										
2		X	X			X	X									
3			X	X			X	X								
4					X	X			X	X						
5						X	X			X	X					
6							X	X			X	X				
7									X	X			X	X		
8										X	X			X	X	
9											X	X			X	X

Figure 6.2: Element – Node information of the example in Figure 6.1

Storing the input data among processors, there are 5 parameters required to index the data each processor has.

1. nsizeiea is 1 plus the number of elements and material set information stored by the processor, or this is the size of $IE^{(r)}$ array.
2. nsizejea is the size of element connectivities stored by the processor, or this is the

size of $JE^{(r)}$ array.

3. noffiejea is the cumulative number of elements owned by the processors that have a rank lower than the processor itself. For example, noffiejea on processor 2 is the summation of elements stored on processor 0 and 1.

4. nsizemynode is the number of nodes stored by the processor.

5. noffmynode is the cumulative number of nodes owned by the processors which has the rank lower than the processor itself. For example, noffmynode on processor 2 is the summation of nodes stored on processor 0 and 1.

From these 5 parameters, sizes of arrays required to store the input data on each processor are given in Table 6.3.

Table 6.3: Sizes of local arrays on each processor

Array	Size
$IE^{(r)}$	nsizeiea
$JE^{(r)}$	nsizejea
$x^{(r)}$	nsizemynode
$y^{(r)}$	nsizemynode
$z^{(r)}$ (for 3D problem)	nsizemynode
$matset^{(r)}$	nsizeiea-1

Using 3 processors to store the input data of the example in Figure 6.1, one has,

On processor 0,

$nsizeiea^{(0)} = 4$

$nsizejea^{(0)} = 12$

$noffiejea^{(0)} = 0$

$nsizemynode^{(0)} = 6$

$noffmynode^{(0)} = 0$

$IE^{(0)} = (1, 5, 9, 13)$

$JE^{(0)} = (1, 5, 6, 2, \mathbf{2, 6, 7, 3,} 3, 7, 8, 4)$

$x^{(0)} = (0.000, 2.000, 4.000, 6.000, 0.000, 2.000)$

$y^{(0)} = (0.000, 0.000, 0.000, 0.000, 2.000, 2.000)$

$matset^{(0)} = (1, 2, 1)$

On processor 1,

$nsizeiea^{(1)} = 4$

$nsizejea^{(1)} = 12$

$noffiejea^{(1)} = 3$

$nsizemynode^{(1)} = 5$

$noffmynode^{(1)} = 6$

$IE^{(1)} = (1, 5, 9, 13)$

$JE^{(1)} = (5, 9, 10, 6, \mathbf{6, 10, 11, 7,} 7, 11, 12, 8)$

$x^{(1)} = (4.000, 6.000, 0.000, 2.000, 4.000)$

$y^{(1)}$ = (2.000, 2.000, 4.000, 4.000, 4.000)

$matset^{(1)}$ = (2, 3, 2)

On processor 2,

$nsizeiea^{(2)}$ = 4

$nsizejea^{(2)}$ = 12

$noffiejea^{(2)}$ = 6

$nsizemynode^{(2)}$ = 5

$noffmynode^{(2)}$ = 11

$IE^{(2)}$ = (1, 5, 9, 13)

$JE^{(2)}$ = (9, 13, 14, 10, **10, 14, 15, 11,** 11, 15, 16, 12)

$x^{(2)}$ = (6.000, 0.000, 2.000, 4.000, 6.000)

$y^{(2)}$ = (4.000, 6.000, 6.000, 6.000, 6.000)

$matset^{(2)}$ = (1, 2, 1)

Although the partitioning of all the input data is discussed in this section, only the partitioning of element connectivity is implemented in this work. The partitioning of node coordinates and material set information will be implemented in the future version of the code.

6.2 Data preparing for ParMETIS to break the domain into subdomains

To partition the entire domain, ParMETIS (Karypis, Schloegel and Kumar) is used to perform the tasks, and the input information ParMETIS requires is the distributed

adjacency structure of the domain. This distributed adjacency structure of the domain, which is used in a parallel computing environment, is extended from the serial adjacency structure of the domain. The serial adjacency structure is described here first. Then, the distributed adjacency structure will be explained later in this section.

The adjacency structure is represented by two arrays, IAKEEP and JA, and stored as follows. The adjacency list of node i is stored in array JA starting from index IAKEEP(i) to IAKEEP(i+1)-1. For example, the adjacency list of node 3 is stored in JA array from index IAKEEP(3) to IAKEEP(4)-1. Hence, the adjacency lists for each node are stored consecutively in the array JA, while the array IAKEEP is used to index the starting point in JA array of each node. It should be noted that the partitioning of the domain in this work is done in node level.

The adjacency arrays, in this work, are obtained from element connectivity information, which is stored in IE and JE arrays. The steps to obtain serial adjacency arrays using one processor could be explained as follows.

1. Find the transpose of element connectivity information and store in IET and JET arrays. In other words, JE array represents the list of nodes attached to each element, and IE array is used to index the starting point of each element in JE array. Meanwhile, JET array represents the list of elements attached to each node, and IET array is used to index the starting point of each node in JET array. Sizes of IE, JE, IET and JET are shown in Table 6.4.

Table 6.4: Sizes of element connectivity arrays and their transpose

Array	Size
IE	number of elements + 1 (nel+1)
JE	IE(nel+1)-1
IET	number of nodes + 1 (node+1)
JET	IET(node+1)-1 = IE(nel+1)-1

2. Set IAKEEP(1) = 1
3. Consider node-by-node in IET array.
4. For the i^{th} node, find the list of elements attached to the node.
5. Consider element-by-element in IE array attached to the i^{th} node.
6. For the j^{th} element attached to the i^{th} node, find the list of nodes attached to that element.
7. Store the list of nodes attached to each element in step 6 (with no duplication) in JA array. At the end, update the index of the starting point of the next node number in array IAKEEP.
8. Repeat step 3 until all nodes in IET array are considered.

For the example in Figure 6.1, the serial adjacency arrays of the domain can be expressed as:

IAKEEP = (1, 4, 9, 14, 17, 22, 30, 38, 43, 48, 56, 64, 69, 72, 77, 82, 85)

JA = (2,5,6, **1,3,5,6,7,** 2,4,6,7,8, **3,7,8,** 1,2,6,9,10, **1,2,3,5,7,9,10,11,**
2,3,4,6,8,10,11,12, **3,4,7,11,12,** 5,6,10,13,14, **5,6,7,9,11,13,14,15,**
6,7,8,10,12,14,15,16, **7,8,11,15,16,** 9,10,14, **9,10,11,13,15,** 10,11,12,14,16,
11,12,15)

As mentioned earlier, the distributed adjacency structure is the extension of the serial adjacency structure. The purpose of this format is to store the adjacency structure among the processors, so a bigger problem could be partitioned by ParMETIS. From the output of the previous section, numbers of nodes handled by processor 0, 1 and 2 are 6, 5 and 5, respectively. Therefore, the serial adjacency structure obtained earlier could be expressed in the form of a distributed adjacency structure as below.

On processor 0,

$IAKEEP^{(0)}$ = (1, 4, 9, 14, 17, 22, 30)

$JA^{(0)}$ = (2,5,6, **1,3,5,6,7,** 2,4,6,7,8, **3,7,8,** 1,2,6,9,10, **1,2,3,5,7,9,10,11)**

On processor 1,

$IAKEEP^{(1)}$ = (1, 9, 14, 19, 27, 35)

$JA^{(1)}$ = (2,3,4,6,8,10,11,12, **3,4,7,11,12,** 5,6,10,13,14, **5,6,7,9,11,13,14,15,**
6,7,8,10,12,14,15,16)

On processor 2,

$IAKEEP^{(2)}$ = (1, 6, 9, 14, 19, 22)

$$JA^{(2)} = (\ 7,8,11,15,16,\ 9,10,14,\ 9,10,11,13,15,\ 10,11,12,14,16,\ 11,12,15)$$

Like the serial adjacency structure, the adjacency of node i is stored in JA array starting from IAKEEP(i) to IAKEEP(i+1)-1. In fact, the domain's IAKEEP and JA arrays have never been constructed, but each processor obtains its own IAKEEP and JA independently. However, there are some communications involved in the procedures since the element connectivity information is distributed among the processors. As a result, the steps to construct the serial adjacency information should be revised as follows in order to acquire the distributed adjacency information.

1. Each processor checks all elements in $JE^{(r)}$ array, creates a list of number of elements attached to each node and stores in $IELCUM^{(r)}$ array. In the example in Figure 6.1, each processor checks its own JE array and creates $IELCUM^{(r)}$ array as follows.

$$ielcum^{(0)} = \begin{Bmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 1 \\ 2 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}, ielcum^{(1)} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 2 \\ 2 \\ 1 \\ 1 \\ 2 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}, ielcum^{(2)} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 2 \\ 2 \\ 1 \\ 1 \\ 2 \\ 2 \\ 1 \end{Bmatrix}$$

Note: $IELCUM_6^{(0)} = 2$ indicates that there are two elements of processor 0

attached to node 6.

2. Then, IELCUM arrays on all processors are combined. Each processor obtains combined IELCUM array as,

$$ielcum^{combined} = \begin{Bmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 2 \\ 4 \\ 4 \\ 2 \\ 2 \\ 4 \\ 4 \\ 2 \\ 1 \\ 2 \\ 2 \\ 1 \end{Bmatrix}$$

The combined ielbum array, basically, represents the number of elements attached to each node. For instance, node 7 is attached by 4 elements, or node 12 is attached by 2 elements. In other words, this information can be used to calculate the size of $JET^{(r)}$ array each processor requires. From the previous section, processor 0, 1 and 2 store the information of 6, 5 and 5 nodes, respectively. Therefore, the sizes of JET of processors 0, 1 and 2 are 12, 16 and 8, respectively.

3. Each processor exchanges element connectivity information. From Figure 6.2, the element-node graph could be partitioned as in Figure 6.3.

Element \ Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	$E_0^{(0)}$						$E_1^{(0)}$					$E_2^{(0)}$				
2																
3																
4	$E_0^{(1)}$						$E_1^{(1)}$					$E_2^{(1)}$				
5																
6																
7	$E_0^{(2)}$						$E_1^{(2)}$					$E_2^{(2)}$				
8																
9																

Figure 6.3: Partitioning of element-node information of the structure in Figure 6.1

For terms $E_i^{(j)}$ in Figure 6.3, subscript i denotes the target processor the element connectivity block is sent to, and superscript j denotes the owner of the element connectivity block. For example, $E_1^{(2)}$ belongs to processor 2 and will be sent to processor 1 during this step. At the end of the step, each processor has node-element information as below.

Processor 0:

IET = (1, 2, 4, 6, 7, 9, 13)

JET = (1, 1, 2, 2, 3, 3, 1, 4, 1, 2, 4, 5)

Processor 1:

IET = (1, 5, 7, 9, 13, 17)

JET = (2, 3, 5, 6, 3, 6, 4, 7, 4, 5, 7, 8, 5, 6, 8, 9)

Processor 2:

IET = (1, 3, 4, 6, 8, 9)

JET = (6, 9, 7, 7, 8, 8, 9)

4. Each processor constructs element connectivity information corresponding to the node-element information each processor holds. Although each processor has distributed node-element information from step 3, the distributed adjacency information still could not be formed, since IE and JE arrays each processor currently holds do not have enough information. Therefore, MYIE and MYJE arrays, which store element-node information corresponding to IET and JET arrays, have to be constructed first. For example, processor 2 should have element-node information of element 6, 7, 8 and 9. In addition to creating MYIE and MYJE arrays, IELIST array is also constructed in order to store the list of elements corresponding to IET and JET arrays. Then, each processor has IELIST, MYIE and MYJE information as below.

Processor 0:

IELIST = (1, 2, 3, 4, 5)

MYIE = (1, 5, 9, 13, 17, 21)

MYJE = (1,5,6,2, **2,6,7,3,** 3,7,8,4, **5,9,10,6,** 6,10,11,7)

Processor 1:

IELIST = (2, 3, 4, 5, 6, 7, 8, 9)

MYIE = (1, 5, 9, 13, 17, 21, 25, 29, 33)

MYJE = (2,6,7,3, **3,7,8,4,** 5,9,10,6, **6,10,11,7,** 7,11,12,8, **9,13,14,10,** 10,14,15,11, **11,15,16,12**)

Processor 2:

IELIST = (6, 7, 8, 9)

MYIE = (1, 5, 9, 13, 17)

MYJE = (7,11,12,8, **9,13,14,10,** 10,14,15,11, **11,15,16,12**)

5. The distributed adjacency information on each processor could be obtained by using the algorithm to find the serial adjacency information where MYIE and MYJE will play the same roles as IE and JE, respectively. After the step is done, each processor has the distributed adjacency information as below.

On processor 0,

IAKEEP(0) = (1, 4, 9, 14, 17, 22, 30)

JA(0) = (2,5,6, **1,3,5,6,7,** 2,4,6,7,8, **3,7,8,** 1,2,6,9,10, **1,2,3,5,7,9,10,11**)

On processor 1,

IAKEEP(1) = (1, 9, 14, 19, 27, 35)

JA(1) = (2,3,4,6,8,10,11,12, **3,4,7,11,12,** 5,6,10,13,14, **5,6,7,9,11,13,14,15,** 6,7,8,10,12,14,15,16)

On processor 2,

IAKEEP(2) = (1, 6, 9, 14, 19, 22)

JA(2) = (7,8,11,15,16, **9,10,14,** 9,10,11,13,15, **10,11,12,14,16,** 11,12,15)

6.3 Post processing of ParMETIS's result to find subdomains' information

To demonstrate all the features in the post-partitioning phase of the domain, a 10-by-10-node of rectangular elements example is introduced in Figure 6.4. In this example, there are 4 degrees of freedom per node and the Dirichlet boundary conditions occur on the first and the third degrees of freedom of nodes 1 to 10. There is no external load

acting on the problem, but the prescribed displacements are introduced on each node.

91	92	93	94	95	96	97	98	99	100
81	82	83	84	85	86	87	88	89	90
71	72	73	74	75	76	77	78	79	80
61	62	63	64	65	66	67	68	69	70
51	52	53	54	55	56	57	58	59	60
41	42	43	44	45	46	47	48	49	50
31	32	33	34	35	36	37	38	39	40
21	22	23	24	25	26	27	28	29	30
11	12	13	14	15	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10

Figure 6.4: 10-by-10-node rectangular elements example

1	1	1	1	1	1	1	1	3	1
1	1	1	1	1	1	1	1	1	3
1	1	1	1	1	1	1	1	3	3
1	2	1	2	1	1	1	3	1	1
2	2	2	1	1	3	3	3	3	3
2	3	2	1	2	2	2	3	3	3
2	3	3	3	2	3	2	3	3	3
2	3	2	2	2	2	3	3	3	3
2	2	2	2	2	2	2	3	3	3
2	2	2	2	2	2	2	3	3	3

Figure 6.5: Node owner after ParMETIS

After the ParMETIS phase, the result from ParMETIS can be shown in Figure 6.5. Although the result from ParMETIS describes the owner of each node, further computations are required to acquire boundary nodes, interior nodes, element connectivities information, Dirichlet boundary conditions, external loads, etc., of each subdomain. The 16 steps post-processes after ParMETIS phase are clearly explained in this section

1. Identify the owner of each element, and the system boundary and interior nodes.

   Before proceeding to the next step, all elements in the domain have to be

assigned the owners so that the edge of the subdomains can be defined. In this work, based on the element connectivity information, the owner of an element is the majority of the nodes' owners of that element. Moreover, the elements are categorized in two types. The first one is interior elements, which are the elements that each of the nodes, excluding boundary nodes, is originally owned by the same subdomain. The second one is boundary elements, which are the elements that each of the nodes, excluding boundary nodes, is originally owned by different processors. Hence, the boundary nodes can be obtained from the boundary elements. The pseudo code of this step is shown below.

Input: (see Appendix A.2 for explanation of each variable)

node, nel, nsub, ndofpn, nsizeiea, nsizejea, noffiejea, MET(node), IE(nsizeiea), JE(nsizejea), nmpcg, IAMPCG(nmpcg+1), JAMPCG(IAMPCG(nmpcg+1)-1)

Output: (see Appendix A.2 for explanation of each variable)

icount – Number of boundary nodes in the entire domain

MET(node), IELMAP(nel), IELMAPMPC(nmpcg), IELOWNER(NEL), IMPCOWNER(nmpcg), NCHK(nsub+2), NCHKMPC(nsub+2)

The master processor performs:

```
NCHK(1:nsub+2) = 0
do irank = 1 to np
    do i = the first element to the last element in the data block
        set idone = 0, NCHKMPC2(1:nsub) = 0, imax = 0, iown = 0
        do j = 1  to number of nodes in the element
            itmp = JE(j) ; node number in global numbering format
            skip to the next node if itmp is a boundary node
            idone = idone+1 ; update number of node in the element excluding
              boundary nodes
```

```
        iref = abs(mod(MET(itmp),nsub)) ; The owner of the node (from
            ParMETIS data)
        NCHKMPC2(iref) = NCHKMPC2(iref)+1 ; update the counter
        if (imax < NCHKMPC2(iref) then
            imax = NCHKMPC2(iref) : update imax
            iown = iref : update the owner of the element
        endif
    enddo j
    if all nodes in the element are boundary nodes, the owner of the first
       node is the owner of the element.
    if imax is equal idone,
        all nodes in the element excluding boundary nodes are belong to the same
            processor;
        NCHK(iown) = NCHK(iown)+1 ; update the counter of interior elements
            of iown^th subdomain
        IELOWNER(i) = iown ; record the owner of i^th element
    else
        i^th element is a boundary element;
        IELOWNER(i) = -iown ; record the owner of i^th element where the minus
            value indicates boundary element
        itmp = nel-NCHK(nsub+1)  ; location of the boundary element to be stored
            in IELMAP array
        NCHK(nsub+1) = NCHK(nsub+1)+1  ; update number of boundary
            elements
        IELMAP(itmp) = i              ; record element ID to IELMAP array
        do j = the first node of the element to the last node of the element
            inode = JE(j)             ; node id
            iref = mod(MET(inode-1),nsub)+1          ; the owner of the node
```

skip j^{th} node if the owner of the node is the same as the owner of the element

skip if inode is a boundary node

icount = icounter+1 ; update the counter of the boundary nodes

MET(inode) = MET(inode)+icount*nsub ; record the ID of the boundary node to MET array

enddo j

endif

enddo i

Receive the data from irankth processor. skip if irank is np.

enddo irank

repeat i loop to find the interior and boundary artificial elements and boundary nodes from MPC equations

convert the format of NCHK such that boundary elements are stored in IELMAP array from NCHK(nsub+1) to (NCHK(nsub+2)-1), and interior elements of subdomain i are stored from NCHK(i) to NCHK(i+1)-1

Also, convert the format of NCHKMPC array such that boundary artificial elements are stored in IELMAPMPC array from NCHKMPC(nsub+1) to (NCHKMPC(nsub+2)-1), and interior artificial elements of subdomain i are stored from NCHKMPC(i) to NCHKMPC(i+1)-1

Using IELOWNER and IMPCOWNER arrays to store the interior elements to IELMAP and IELMAPMPC, respectively

Sending number of all boundary nodes, MET, NCHK, IELMAP, NCHKMPC, IELMAPMPC, IELOWNER and IMPCOWNER to the other processors.

The slave processors perform:

Sending element connectivities information to the master processor.

Receiving number of all boundary nodes, MET, NCHK, IELMAP, NCHKMPC, IELMAPMPC, IELOWNER and IMPCOWNER to the other processors.

After step 1, all processors have the following information.

MET(:) = 2 2 2 2 2 2 2 9 3 3 2 2 2 2 2 2 2 6 3 3 2 12 23 26 2 2 15 18 3 3 2 21 42
30 2 33 2 36 3 3 2 39 2 43 53 59 62 48 3 3 2 2 2 49 1 57 75 3 3 3 64 86 67 71
1 1 76 90 79 82 1 1 1 1 1 1 1 1 93 3 1 1 1 1 1 1 1 1 94 102 1 1 1 1 1 1 1 1 99 1

NCHK(:) = 1 22 39 51 82

IELMAP(:) = 50 56 57 58 59 60 64 65 66 67 68 69 70 71 73 74 75 76 77 78 79
1 2 3 4 5 6 11 12 13 14 24 31 32 33 37 38 48 8 9 17 18 26 27 35 36 43 44 45
63 81 80 72 62 61 55 54 53 52 51 49 47 46 42 41 40 39 34 30 29 28 25 23 22
21 20 19 16 15 10 7

IELOWNER(:) = 2 2 2 2 2 2 -2 3 3 -2 2 2 2 2 -2 -2 3 3 -2 -3 -3 -2 -2 2 -2 3 3 -2 -
2 -2 2 2 2 -2 3 3 2 2 -2 -1 -1 -3 3 3 3 -2 -2 2 -1 1 -1 -3 -3 -3 -1 1 1 1 1 1 -1 -1
3 1 1 1 1 1 1 1 1 -3 1 1 1 1 1 1 1 -1 -1

MET and IELOWNER arrays could also be illustrated as in Figure 6.6 and 6.7, respectively.

1	1	1	1	1	1	1	1	99	1
1	1	1	1	1	1	1	1	94	102
1	1	1	1	1	1	1	1	93	3
64	86	67	71	1	1	76	90	79	82
2	2	2	49	1	57	75	3	3	3
2	39	2	43	53	59	62	48	3	3
2	21	42	30	2	33	2	36	3	3
2	12	23	26	2	2	15	18	3	3
2	2	2	2	2	2	2	6	3	3
2	2	2	2	2	2	2	9	3	3

Figure 6.6: MET array after step 1

1	1	1	1	1	1	1	-1	-1
1	1	1	1	1	1	1	1	-3
-1	1	1	1	1	1	-1	-1	3
-2	-2	2	-1	1	-1	-3	-3	-3
2	2	-2	-1	-1	-3	3	3	3
-2	-2	-2	2	2	2	-2	3	3
-2	-3	-3	-2	-2	2	-2	3	3
-2	2	2	2	2	-2	-2	3	3
2	2	2	2	2	2	-2	3	3

Figure 6.7: Elements' owner after step 1

In Figure 6.6, the value at each node indicates the MET value of the node. When the value at the node is less than or equal to the number of subdomains, it means that the node is an interior node, and the value at the node is the subdomain the node belongs to. On the other hand, when the value at the node is more than the number of subdomains, it indicates that the node is a boundary node. Moreover, the value at the node will tell the global boundary node id and the subdomain the node belongs to. The global boundary node id of i^{th} boundary node is (MET(i)-1)/nsub. Then, the subdomain the i^{th} node belongs to is mod(MET(i)-1,nsub)+1.

In Figure 6.7, the value at each element represents the subdomain the element belongs to, and the minus sign indicates the boundary element.

2. Each processor utilizes MET array information to find the number and the list of interior dofs of each subdomain and store in NCHK2 and IA, respectively. NCHK2(i) represents the starting location of interior nodes in IA array of subdomain i.

   nchk2 = 1 29 54 68 (subdomain 2 has 54-29 = 25 interior dofs)

   IA = 55 65 66 71 72 73 74 75 76 77 78 81 82 83 84 85 86 87 88 91 92 93 94 95
   96 97 98 100 1 2 3 4 5 6 7 11 12 13 14 15 16 17 21 25 26 31 35 37 41 43 51
   52 53 9 10 19 20 29 30 39 40 49 50 58 59 60 80

3. Each processor obtains element connectivities information corresponding to interior elements and boundary elements information in IELMAP array. This step is required in order to have the processors store only the necessary information instead of the whole element connectivities information. The pseudo code in this step is given below.

```
MYELLIST(1:nel) = 0
do i = the first to the last boundary element
     iel = IELMAP(i)   ; element ID
     MYELLIST(iel) = 1        ; iel^th element belongs to the subdomain
enddo
do i = the first to the last interior element
     iel = IELMAP(i)   ; element ID
     MYELLIST(iel) = 1        ; iel^th element belongs to the subdomain
enddo
```

If any of its own element connectivity information associated with the subdomain, do

- index the id of the element to MYELLIST array
- copy JE information of that element to subdomain's MYJE
- update MYIE array to index the starting location of the element in MYJE array

Receive distributed element connectivities information from other processors and perform the previous step again until all the connectivities information has been checked.

4. Each processor creates the list of its own boundary nodes. All the elements, both boundary and interior elements, are considered. For each interior element, if any node in the element is a system boundary node, it will also be a boundary node of the subdomain. For a boundary element that belongs to the subdomain, all nodes in the element that do not belong to the subdomain are boundary nodes. If a boundary element does not belong to the subdomain, the nodes in that element that belong to the subdomain are also the boundary nodes of the subdomain. The pseudo code of this step could be written as below.

```
For ISUB^th subdomain,
ncbd = 0
do i = the first to the last interior element
    do j = the first node to the last node of the i^th element
        skip to the next node if MET(j^th node) is less than or equal number of
            subdomains since the node is either already counted or it is an interior
            node
        ncbd = ncbd+1        ; update counter of subdomain's boundary nodes
        IBOUND(ncbd) = j^th node    ; record the boundary node to IBOUND array
        MET(j^th node) = -1*MET(j^th node)    ; marked as an already counted node
    enddo
```

enddo

do i = the first to the last boundary element

iowner = -1*ielowner(i) ; the owner of the element

if iowner = ISUB, the element belongs to the subdomains

do j = the first node to the last node of the all nodes in the element

skip to the next node if MET(j^{th} node) is less than zero since the node is already counted

if node j is a boundary node, node j is the boundary node in the subdomain. Then,

ncbd = ncbd+1 ; update counter of subdomain's boundary nodes

IBOUND(ncbd) = j^{th} node ; record the boundary node to IBOUND array

MET(j^{th} node) = -1*MET(j^{th} node) ; marked as an already counted node

enddo

elseif iowner not equal to ISUB, the element does not belong to the subdomain.

do j = all nodes in the element

skip to the next node if MET(j^{th} node) is less than zero since the node is already counted

if j^{th} node belongs to the subdomain, the node is the boundary node. Then

ncbd = ncbd+1 ; update counter of subdomain's boundary nodes

IBOUND(ncbd) = j^{th} node ; record the boundary node to IBOUND array

MET(j^{th} node) = -1*MET(j^{th} node) ; marked as an already counted node

enddo

endif

enddo

Redo the steps in this phase with artificial elements from MPC equations

nodes = ncbd+(NCHK(ISUB+1)-NCHK(ISUB))

After step 4, each processor has:

Processor 0:

Number of nodes in subdomain : 46

Number of boundary nodes : 18

IBOUND(1:18) : 56 62 63 64 67 89 79 44 54 45 46 61 57 69 70 68 99 90

Processor 1:

Number of nodes in subdomain : 48

Number of boundary nodes : 23

IBOUND(1:23) : 22 23 24 36 27 34 44 45 46 47 42 63 64 54 18 8 28 32 38 33 48 61 62

Processor 2:

Number of nodes in subdomain : 40

Number of boundary nodes : 26

IBOUND(1:26) : 8 18 28 38 48 47 57 69 79 70 22 27 32 33 23 34 24 36 42 56 46 67 68 89 90 99

The local boundary node ID of the subdomains could be written in Figure 6.8. Since the boundary node ID on one subdomain may be different from another subdomain, the boundary node ID of subdomains 1, 2, 3 and the original MET value are located on the top right, lower left, lower right and top left of the node,

respectively.

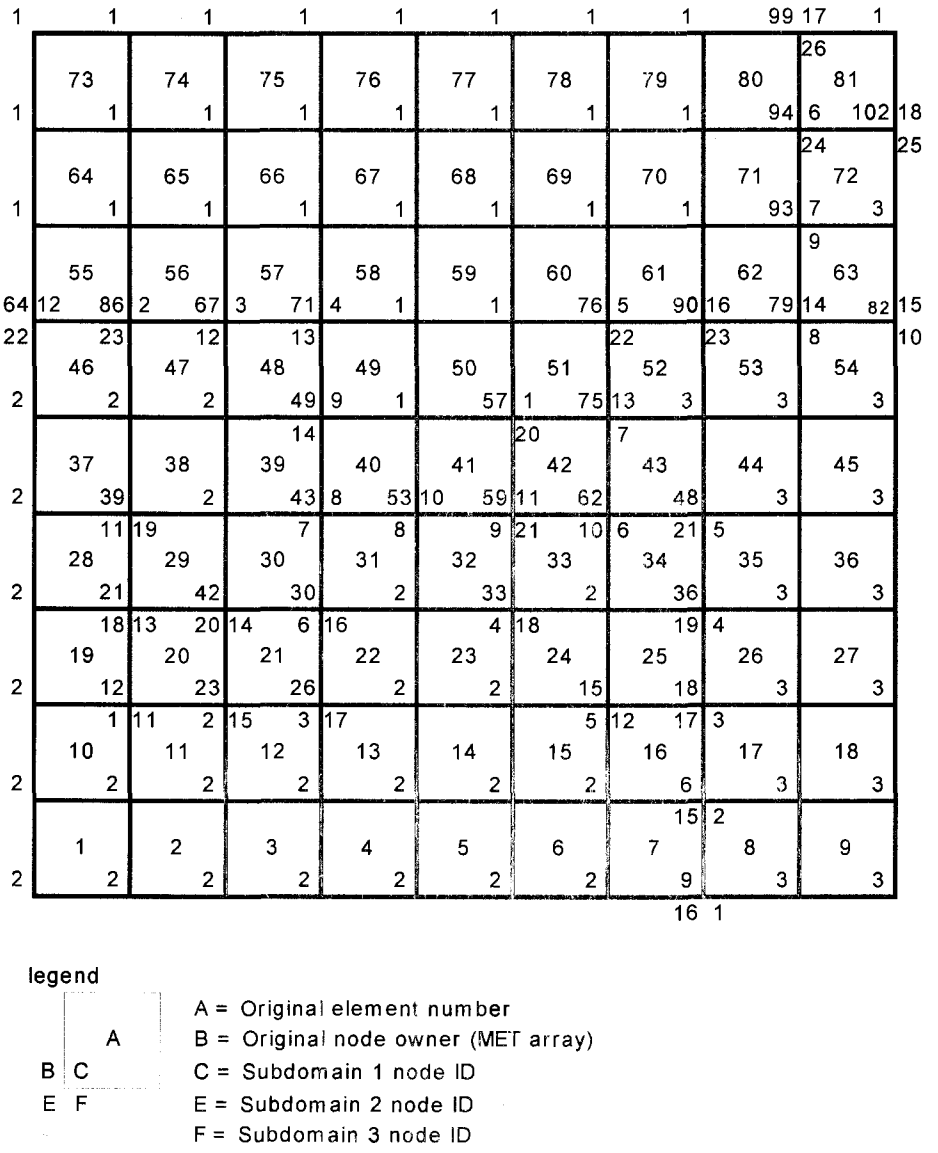


Figure 6.8: Local boundary node ID of the subdomains

5. Find the association between local boundary degrees of freedom and global boundary degrees of freedom. Using MET array information to construct IBDOFARE of size nbdof that maps between local boundary degrees of freedom and global boundary degrees of freedom. The output information is stored in

IBDOFARE array, where IBDOFARE(local boundary degree of freedom) returns global boundary degree of freedom ID. The pseudo code of this step is included below.

```
do i = 1 to number of subdomain's boundary nodes
    inode = glocal node id of i^th node
    Find the global boundary node id from id = met(inode)/nsub
    if mod(met(inode),nsub) is zero id = id-1  ;  correct the global boundary node id
    IBDOFARE(i) = id
enddo
ITEMP(1:nbdof) = IBDOFARE(1:nbdof)
do i = 1 to number of subdomain's boundary nodes
    indx = (ITEMP(i)-1)*ndofpn        ! the index location in ITEMP
    ilocat = (i-1)*ndofpn             ! the index location in IBDOFARE
    do j = 1 to ndofpn
        IBDOFARE(ilocat+j) = indx+j   ! store global dof to IBDOFARE array
    enddo
enddo
```

Upon the step is done, each processor has IBDOFARE array information as below.

Processor 0:

IBDOFARE = 18 28 22 23 25 31 30 14 16 17 19 21 24 26 27 29 32 33

Processor 1:

IBDOFARE = 3 7 8 10 4 9 14 17 19 20 12 22 23 16 1 2 5 6 11 13 15 21 28

Processor 2:

IBDOFARE = 2 1 5 11 15 20 24 26 30 27 3 4 6 13 7 9 8 10 12 18 19 25 29 31 33 32

6. Construct the association of the subdomain's nodes with global nodes. During this phase, a temporary IAKEEP array of size number of total nodes of the domain is created in order to easily identify which nodes belong to the subdomain. If IAKEEP(i) is 0, the ith node does not belong to the subdomain. On the other hand, the value of IAKEEP(i), if not zero, tells the subdomain's node ID of ith node.

```
ncounter = 0
do i = 1 to number of subdomain's boundary nodes
    node = global node id associated with ith subdomain's boundary node
    ncounter = ncounter+1
    IAKEEP(node) = ncounter
enddo
do i = the first to the last of subdomain's interior nodes
    node = global node id associated with ith subdomain's interior node
    ncounter = ncounter+1
    IAKEEP(node) = ncounter
enddo
```

Finally, each processor obtains;

Processor 0:

IAKEEP = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 10 11 0 0 0 0 0 0 0 9 19 1 13 0 0 0 12 2 3 4 20 21 5 16 14 15 22 23 24 25 26 27 28 29 7 0 30 31 32 33 34 35 36 37 6 18 38 39 40 41 42 43 44 45 17 46

Processor 1:

IAKEEP = 24 25 26 27 28 29 30 16 0 0 31 32 33 34 35 36 37 15 0 0 38 1 2
3 39 40 5 17 0 0 41 18 20 6 42 4 43 19 0 0 44 11 45 7 8 9 10 21 0 0 46
47 48 14 0 0 0 0 0 0 22 23 12 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Processor 2:

IAKEEP = 0 0 0 0 0 0 0 1 27 28 0 0 0 0 0 0 0 2 29 30 0 11 15 17 0 0 12 3
31 32 0 13 14 16 0 18 0 4 33 34 0 19 0 0 0 21 6 5 35 36 0 0 0 0 0 20 7
37 38 39 0 0 0 0 0 0 22 23 8 10 0 0 0 0 0 0 0 0 0 9 40 0 0 0 0 0 0 0 0 24
25 0 0 0 0 0 0 0 0 26 0

The nodes ID of each subdomain could be expressed in Figure 6.9. Please note that, at each particular node, the original node owner from the value of MET array is shown in the top left of the node. The node ids of the node in subdomain 1, 2 and 3 are shown in the top right, lower left and lower right of the node, respectively.

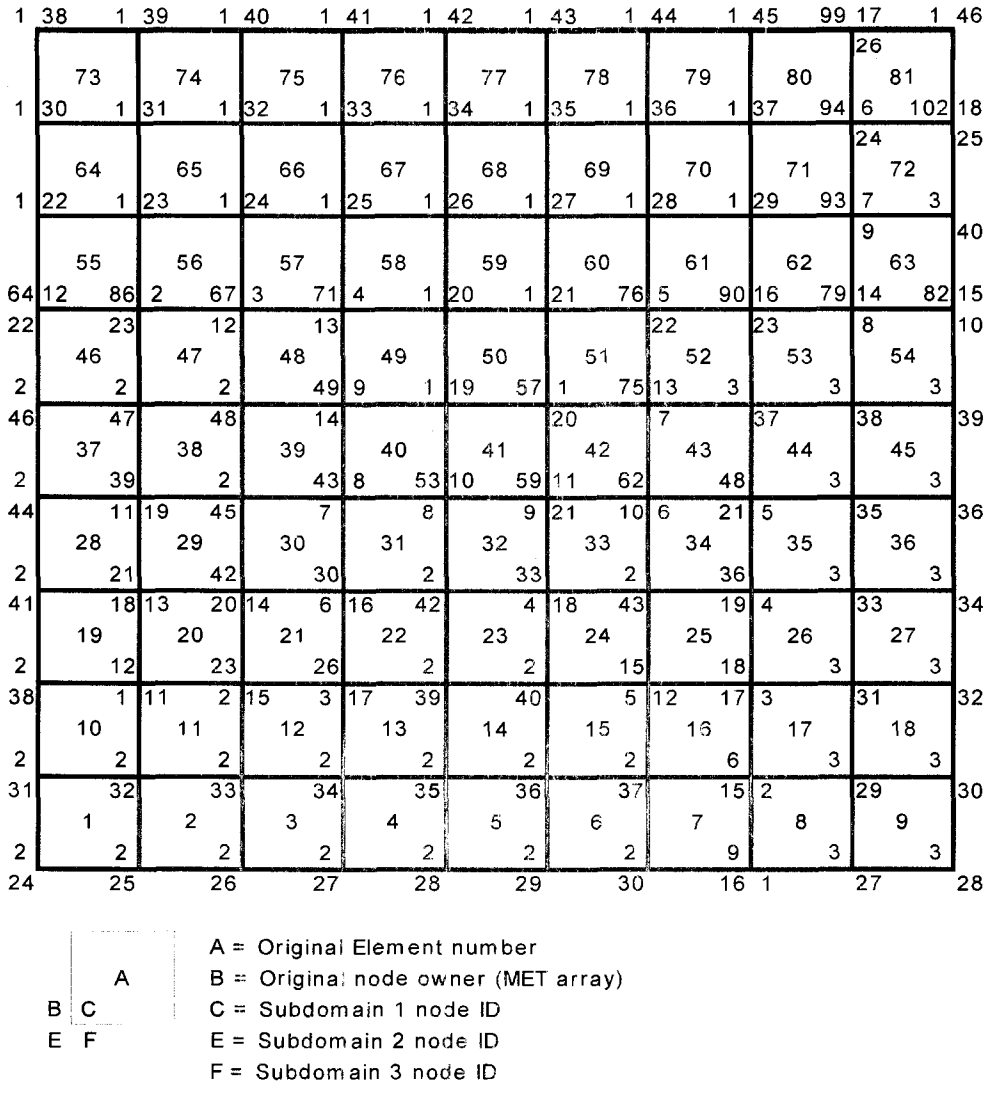


Figure 6.9: Local node ID of the subdomains

7. Partitioning Dirichlet boundary conditions to each subdomain. Each processor checks whether the Dirichlet boundary conditions take place in its subdomain. Then, the Dirichlet boundary conditions that occurred in the subdomain are incorporated into the subdomain data. The pseudo code in this part will be;

```
set ndir = 0 ; ndir is the number of subdomain's Dirichlet boundary conditions
do i = 1 to the number of global Dirichlet boundary conditions
    node = global node id associated with ith Dirichlet boundary condition dof
```

check if node in is the subdomain (from the value of IAKEEP(node))

skip if the node is not in the subdomain

ndir = ndir+1 ; update number of subdomain's Dirichlet boundary conditions

convert local node id to local dof id

NBCDOFS(ndir) = local dof id

enddo

Each processor obtains this information after stop 7;

Processor 0:

Number of Dirichlet Boundary conditions : 0

Processor 1:

Number of Dirichlet Boundary conditions : 16

NBCDOFS : 93 95 97 99 101 103 105 107 109 111 113 115 117 119 61 63

Processor 2:

Number of Dirichlet Boundary conditions : 6

NBCDOFS : 1 3 105 107 109 111

8. Find the number of interior and boundary elements of each element type in each subdomain. Then, the list of number of elements of each type and the list of regular and special elements are created.

set niel(1:neltype) = 0 ; niel is the number of interior elements of each element type where niel(i) is the number of interior elements of i^{th} element type

set nbel(1:neltype) = 0 ; nbel is the number of boundary elements of each element type where nbel(i) is the number of boundary elements of i^{th} element type

```
do it = 1 to number of element types
    do i = the first to the last interior elements of it^th element type
        skip if i^th element is not of type it^th element
        update niel(it) = niel(it)+1
    enddo
    do i = the first to the last boundary elements of it^th element type
        skip if i^th element is not of type it^th element
        skip if the element does not belong to the subdomain
        update nbel(it) = nbel(it)+1
        record the i^th special element to ITEMP array
    enddo
enddo
nels = niel(1:neltype)+nbel(1:neltype) ; nels is the number of subdomain's elements
ncums(1) = 1 ; ncums(i+1)-ncums(i) indicates the number of elements of i^th element
   type
do i = 1 to number of element types
    ncums(i+1) =ncums(i)+niel(i)+nbel(i)
enddo
```

Note: the element type of the example in Figure 6.4 is of type 6.

At the end of this phase,

Processor 0:

No. of regular elements of each element type: 0 0 0 0 0 21

No. of special elements of each element type: 0 0 0 0 0 9

ncums = 1 1 1 1 1 1 31

No. of sub elements: 30

Processor 1:

No. of interior elements: 0 0 0 0 0 17

No. of boundary elements: 0 0 0 0 0 15

ncums = 1 1 1 1 1 1 33

No. of sub elements: 32

Processor 2:

No. of interior elements: 0 0 0 0 0 12

No. of boundary elements: 0 0 0 0 0 7

ncums = 1 1 1 1 1 1 20

No. of sub elements: 19

9. Create element connectivity for each subdomain. In this step, each processor considers its own interior elements and boundary elements. Then, the element connectivity information is stored in IES and JES arrays. Also, IBDCHK array of size ncbd+1, number of subdomain's boundary node plus 1, is constructed in order to find the association of subdomain's boundary nodes to the subdomain. After all the previous steps are done, there might be a case that a subdomain's boundary node does not connect to any element in the subdomain at all. For example, local boundary nodes 12, 18, 19 and 26 of subdomain 3 do not connect to any element

of subdomain 3 at all. As a result, such boundary nodes have to be eliminated from the subdomain. The pseudo code in this step is given below.

```
location = 0, nelcount = 1, IES(1) = 1, IBDCHK(1:ncbd+1)
do it = 1 to number of element types
    do i = the first to the last subdomain's interior element
        add the list of nodes of i^th element to the end of JES
        if any of node in i^th element is a boundary node, update IBDCHK at the
            location of that node to be 1
        IES(i+1) = IES(i)+(number of nodes in i^th element) ; update the starting index
            of (i+1)^th element
    enddo
    do i = the first to the last subdomain's boundary element
        skip if the element does not belong to the subdomain
        add the list of nodes of i^th element to the end of JES
        if any node in i^th element is a boundary node, update IBDCHK at the location
            of that node to be 1
        IES(i+1) = IES(i)+(number of nodes in i^th element) ; update the starting index
            of (i+1)^th element
    enddo
enddo
```

After the step is done, each processor obtains

Processor 0:

IES = 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101 105 109 113 117

JES = 18 19 20 1 2 22 23 3 3 23 24 4 4 24 25 19 19 25 26 20 20 26 27 5

21 29 30 22 22 30 31 23 23 31 32 24 24 32 33 25 25 33 34 26 26 34 35
27 27 35 36 28 28 36 6 7 29 37 38 30 30 38 39 31 31 39 40 32 32 40 41
33 33 41 42 34 34 42 43 35 35 43 44 36 8 9 18 10 10 18 1 11 9 4 19 18
1 20 5 13 12 21 22 2 5 27 28 15 15 28 7 14 36 44 16 6 6 16 45 17

IBDCHK = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1

Processor 1:

IES = 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89
93 97 101 105 109 113 117 121 125

JES = 24 31 32 25 25 32 33 26 26 33 34 27 27 34 35 28 28 35 36 29 29 36
37 30 32 1 2 33 33 2 3 34 34 3 39 35 35 39 40 36 40 4 43 5 6 7 8 42 42
8 9 4 4 9 10 43 44 46 47 11 11 47 48 45 48 12 13 14 30 37 15 16 31 38
1 32 36 40 5 37 37 5 17 15 38 41 18 1 3 6 42 39 39 42 4 40 5 43 19 17
41 44 11 18 18 11 45 20 20 45 7 6 43 10 21 19 45 48 14 7 46 22 23 47
47 23 12 48

IBDCHK = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Processor 2:

IES = 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73

JES = 1 2 25 23 23 25 26 24 2 3 27 25 25 27 28 26 3 4 29 27 27 29 30 28
4 5 31 29 29 31 32 30 6 7 33 5 5 33 34 31 31 34 35 32 8 9 36 10 11 12
13 14 14 13 15 16 18 17 7 6 7 19 20 33 33 20 8 34 34 8 10 35 9 21 22

36

IBDCHK = 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 1 1 0

Note: the value of IBDCHK array represents whether the boundary nodes are really associated with the subdomain. If IBDCHK(i) is 1, the i^{th} boundary node is connected to the subdomain. On the other hand, the i^{th} boundary node is not connected to the subdomain if IBDCHK(i) is 0. Before performing the next step, IBDCHK array is transformed to a new format such that;

When i goes from 1 to the number of subdomain's boundary nodes,

IBDCHK(i) = 0 describes no association of node i to the subdomain, or

IBDCHK(i) = k describes (k-1) boundary nodes to be eliminated before i^{th} node.

Also, IBDCHK(ncbd+1) describes the total number of eliminated boundary nodes in the subdomain.

Therefore, IBDCHK array could be reformatted as;

Processor 0:

IBDCHK = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 2 2 2 1

Processor 1:

IBDCHK = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

Processor 2:

IBDCHK = 1 1 1 1 1 1 1 1 1 1 1 0 2 2 2 2 2 0 0 4 4 4 4 4 4 0 4

10. Eliminate boundary nodes not associated with the subdomain. From the previous step, IBDCHK array is used to update JES, IAKEEP, IBDOFARE and NBCDOFS arrays. In other words, the ids of some nodes in the subdomain are shifted since some boundary nodes not associated with the subdomain are eliminated. The pseudo code of this part could be written below.

```
nelim = IBCCHK(ncbd+1)      ; number of eliminated boundary nodes
do i = 1 to the last location of JES
    inode = JES(i)          ; node at i^th location of JES array
    if inode is an interior node, JES(i) = inode-nelim
    if inode is a boundary node, JES(i) = inode-(ibdchk(inode)-1)
enddo
do i = 1 to number of domain's nodes
    itemp = iakeep(i) ; old local node id
    skip to the next node if i^th node is not in the subdomain
    reset IAKEEP(i) = 0 if the node is eliminated
    if the node is an interior node, IAKEEP(i) = itemp-nelim
    if the node is a boundary node, IAKEEP(i) = itemp-(ibdchk(inode)-1)
enddo
ncount = 0    ; counter for new subdomain's boundary nodes
do i = 1 to ncbd
    skip to the next boundary node if the i^th boundary node gets eliminated
    ncount = ncount+1
    IBDOFARE(ncount) = IBDOFARE(i)
enddo
ncbd = ncount; new number of subdomain's boundary nodes
```

```
nelimdof = nelim*ndofpn        ; number of eliminated degrees of freedom
ncount = 0
do i = 1 to number of subdomain's Dirichlet boundary conditions
    idof = NBCDOFS(i)          ; location of i^th subdomain's Dirichlet boundary condition
    inode is the node id corresponding to idof
    if inode is an interior node,
        ncount = ncount+1
        NBCDOFS(ncount) = idof-nelimdof
    if inode is a boundary node,
        ichk = ibdchk(inode)
        skip to the next i if the inode is eliminated
        ncount = ncount+1
        NBCDOFS(ncount) = idof-(ichk-1)*ndofpn
enddo
ndir = ncount
```

Each processor updates JES, IAKEEP, IBDOFARE and NBCDOFS arrays as follows.

Processor 0:

updated JES = 18 19 20 1 2 22 23 3 3 23 24 4 4 24 25 19 19 25 26 20 20
26 27 5 21 29 30 22 22 30 31 23 23 31 32 24 24 32 33 25 25 33 34 26
26 34 35 27 27 35 36 28 28 36 6 7 29 37 38 30 30 38 39 31 31 39 40 32
32 40 41 33 33 41 42 34 34 42 43 35 35 43 44 36 8 9 18 10 10 18 1 11
9 4 19 18 1 20 5 13 12 21 22 2 5 27 28 15 15 28 7 14 36 44 16 6 6 16
45 17

updated IAKEEP = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 10 11 0 0 0 0 0 0 0 9 18 1 13 0 0 0 12 2 3 4 19 20 5 15 14 0 21 22 23 24 25 26 27 28 7 0 29 30 31 32 33 34 35 36 6 17 37 38 39 40 41 42 43 44 16 45

updated ncbd = 17

updated IBDOFARE = 18 28 22 23 25 31 30 14 16 17 19 21 24 26 29 32 33

No Dirichlet boundary condition attaches to the subdomain

Processor 1:

updated JES = 24 31 32 25 25 32 33 26 26 33 34 27 27 34 35 28 28 35 36 29 29 36 37 30 32 1 2 33 33 2 3 34 34 3 39 35 35 39 40 36 40 4 43 5 6 7 8 42 42 8 9 4 4 9 10 43 44 46 47 11 11 47 48 45 48 12 13 14 30 37 15 16 31 38 1 32 36 40 5 37 37 5 17 15 38 41 18 1 3 6 42 39 39 42 4 40 5 43 19 17 41 44 11 18 18 11 45 20 20 45 7 6 43 10 21 19 45 48 14 7 46 22 23 47 47 23 12 48

updated IAKEEP = 24 25 26 27 28 29 30 16 0 0 31 32 33 34 35 36 37 15 0 0 38 1 2 3 39 40 5 17 0 0 41 18 20 6 42 4 43 19 0 0 44 11 45 7 8 9 10 21 0 0 46 47 48 14 0 0 0 0 0 0 22 23 12 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

updated ncbd = 23

updated IBDOFARE = 3 7 8 10 4 9 14 17 19 20 12 22 23 16 1 2 5 6 11 13 15 21 28

updated ndir = 16

updated NBCDOFS = 93 95 97 99 101 103 105 107 109 111 113 115 117 119 61 63

Processor 2:

updated JES = 1 2 25 23 23 25 26 24 2 3 27 25 25 27 28 26 3 4 29 27 27 29 30 28 4 5 31 29 29 31 32 30 6 7 33 5 5 33 34 31 31 34 35 32 8 9 36 10 11 12 13 14 14 13 15 16 18 17 7 6 7 19 20 33 33 20 8 34 34 8 10 35 9 21 22 36

updated IAKEEP = 0 0 0 0 0 0 0 1 23 24 0 0 0 0 0 0 0 2 25 26 0 11 14 16 0 0 0 3 27 28 0 12 13 15 0 0 0 4 29 30 0 0 0 0 0 18 6 5 31 32 0 0 0 0 0 17 7 33 34 35 0 0 0 0 0 0 19 20 8 10 0 0 0 0 0 0 0 0 9 36 0 0 0 0 0 0 0 0 21 22 0 0 0 0 0 0 0 0 0 0

updated ncbd = 22

updated IBDOFARE = 2 1 5 11 15 20 24 26 30 27 3 6 13 7 9 8 18 19 25 29 31 33

updated ndir = 6

updated NBCDOFS = 1 3 89 91 93 95

The new local node id of each subdomain can be represented in Figure 6.10. As

clearly seen, the boundary nodes in each subdomain that do not connect with any element in the subdomain are eliminated from the subdomain.

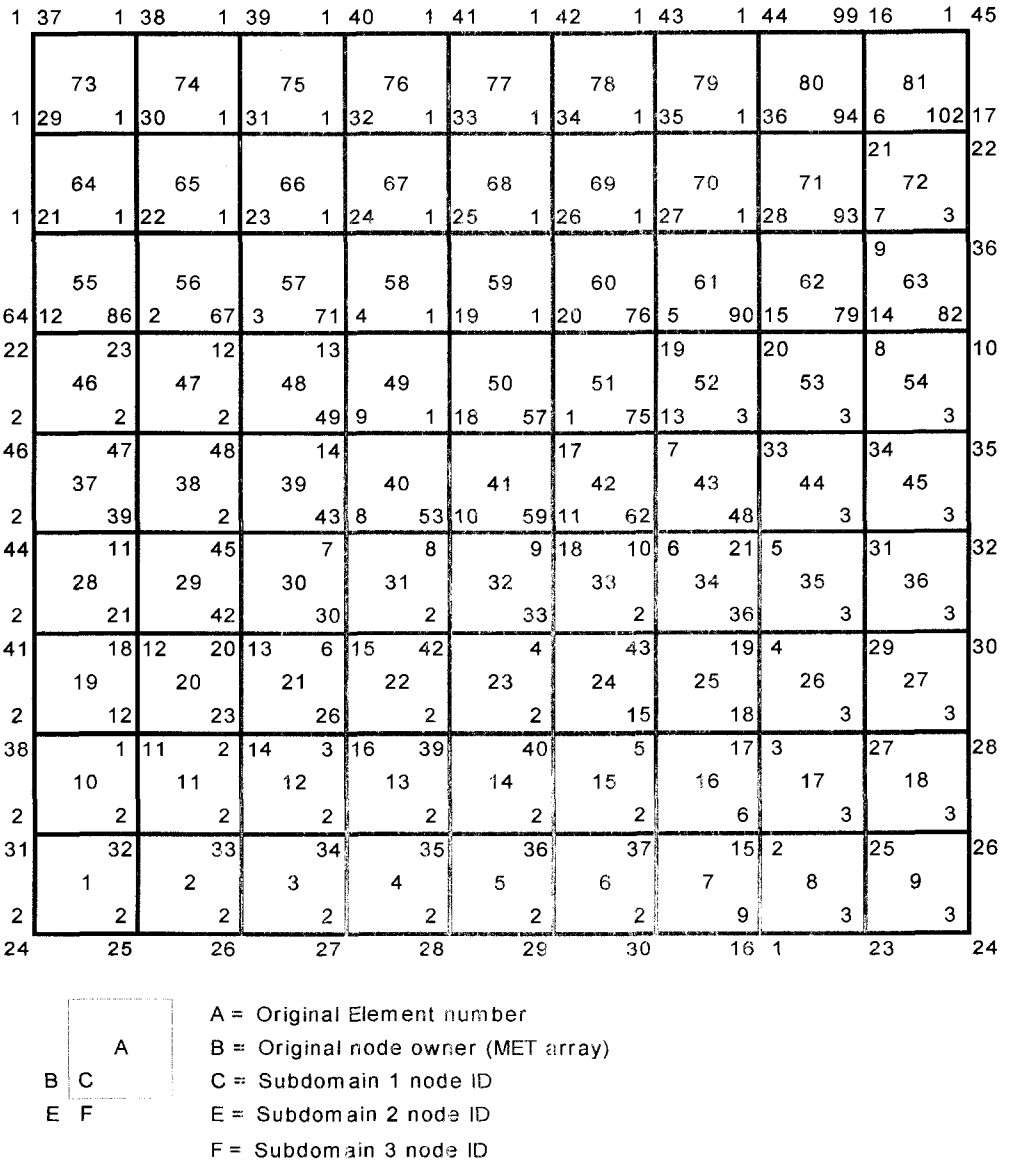


Figure 6.10: New local node ID after elimination of boundary nodes

11. Distributed external loads among subdomains. When the external load occurs on a boundary node, it will be equally distributed to all the subdomains attached to that

boundary node. For example, if a 15 kips external load occurs on boundary node 17, and if there are 3 subdomains attached to that boundary node, the external load on this boundary node of each subdomain will be 5 kips. The pseudo code is given below.

Each processor performs

initialize DIST array of size ngbjdof.

DIST(i) = 1.0 if the subdomain has association with i^{th} global boundary node

DIST(i) = 0.0 if the subdomain has no association with i^{th} global boundary node

Each processor sends and receives DIST array to/ from the other processors and add them up to have the total DIST array.

nloadof = 0 ; number of external loads in the subdomain

do i = 1 to number of domain's external loads

idof is the dof id where the i^{th} external load occurs

inode is the node id where the i^{th} external load occurs

skip if the external load does not occur in the subdomain

nloadof = nloadof+1

ildof is the local dof id where the i^{th} external load occurs

LOADOFS(nloadof) = ildof

if ildof is an interior dof, FF(nloadof) = the value of external load

if ildof is a boundary dof, FF(nloadof) = the value of external load/DIST(inode)

enddo

DIST array on each processor is shown below.

Processor 0:

DIST = 2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0

3.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 1.0 2.0

Processor 1:

DIST = 2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0

3.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 1.0 2.0

Processor 2:

DIST = 2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0

3.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0 2.0 2.0 2.0 2.0 1.0 2.0

12. Create the mapping of subdomain nodes and global nodes. NGBMAP array of size subdomain's nodes is constructed during the step in order to index the local and global node id. At the end, NGBMAP(local node id) returns a global node id. The pseudo code can be written as;

```
do i = 1 to number of total nodes
    id = iakeep(i)      ; local node id
    skip if id is zero (i.e. i^th node is not in the subdomain)
    NGBMAP(id) = i
enddo
```

Then, the NGBMAP array on each processor is:

Processor 0:

NGBMAP = 56 62 63 64 67 89 79 44 54 45 46 61 57 69 68 99 90 55 65 66

71 72 73 74 75 76 77 78 81 82 83 84 85 86 87 88 91 92 93 94 95 96 97

98 100

Processor 1:

NGBMAP = 22 23 24 36 27 34 44 45 46 47 42 63 64 54 18 8 28 32 38 33 48 61 62 1 2 3 4 5 6 7 11 12 13 14 15 16 17 21 25 26 31 35 37 41 43 51 52 53

Processor 2:

NGBMAP = 8 18 28 38 48 47 57 69 79 70 22 32 33 23 34 24 56 46 67 68 89 90 9 10 19 20 29 30 39 40 49 50 58 59 60 80

13. Partitioning MPC equations to appropriate subdomains. From the previous steps, artificial elements from MPC equations are created in order to have all degrees of freedom in an MPC equation belong to the same subdomain. This is done to avoid the coupling between interior degrees of freedom of two or more subdomains. The subdomain's MPC equations information could be written as the pseudo code below.

```
IAMPC(1) = 1
nmpc = 0
njacount = 0
do i = 1 to the number of total MPC equations
    do j = 1 to the number of terms in i^th MPC equation
        idof is the dof corresponding to j^th term of i^th MPC equation
        inode is the node corresponding to j^th term of i^th MPC equation
        skip to the next equation if the node is not in the subdomain
        njacount = njacount+1
        ildof is the local dof corresponding to j^th term of i^th MPC equation
        JAMPC(njacount) = ildof
```

CMPC(njacount) = the coefficient of j^{th} term of i^{th} MPC equation

enddo

nmpc = nmpc+1

iampc(nmpc+1) = iampc(nmpc)+number of terms in i^{th} MPC equation

rmpc(nmpc) = the right-hand-side of i^{th} MPC equation

enddo

14. Construct element nodes list from JES array. This step is done in order to prepare the input data for the next phase. Depending on the number of nodes per element, the output from this step could be NODE1, NODE2, NODE3, NODE4, NODE5, NODE6, NODE7 and/or NODE8. The pseudo code of this step can be expressed below.

id is the rank of node in the element (1^{st}, 2^{nd}, 3^{rd}, ..., i^{th} node of the element)

NODE(nels) represents the id^{th} node of the elements.

iref = id

do i = 1 to number of elements in the suddomains

NODE(i) = JES(iref)

iref = iref+npe

enddo

For the output of the current example, the outputs are NODE1, NODE2, NODE3 and NODE4 since there are 4 nodes per element for this example.

Processor 0:

NODE1 = 18 2 3 4 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 8 10 9 1 12 5 15 36 6

NODE2 = 19 22 23 24 25 26 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 9 18 4 20 21 27 28 44 16

NODE3 = 20 23 24 25 26 27 30 31 32 33 34 35 36 6 38 39 40 41 42 43 44 18 1 19 5 22 28 7 16 45

NODE4 = 1 3 4 19 20 5 22 23 24 25 26 27 28 7 30 31 32 33 34 35 36 10 11 18 13 2 15 14 6 17

Processor 1:

NODE1 = 24 25 26 27 28 29 32 33 34 35 40 6 42 4 44 11 48 30 31 36 37 38 3 39 5 41 18 20 43 45 46 47

NODE2 = 31 32 33 34 35 36 1 2 3 39 4 7 8 9 46 47 12 37 38 40 5 41 6 42 43 44 11 45 10 48 22 23

NODE3 = 32 33 34 35 36 37 2 3 39 40 43 8 9 10 47 48 13 15 1 5 17 18 42 4 19 11 45 7 21 14 23 12

NODE4 = 25 26 27 28 29 30 33 34 35 36 5 42 4 43 11 45 14 16 32 37 15 1 39 40 17 18 20 6 19 7 47 48

Processor 2:

NODE1 = 1 23 2 25 3 27 4 29 6 5 31 8 11 14 18 7 33 34 9

NODE2 = 2 25 3 27 4 29 5 31 7 33 34 9 12 13 17 19 20 8 21

NODE3 = 25 26 27 28 29 30 31 32 33 34 35 36 13 15 7 20 8 10 22

NODE4 = 23 24 25 26 27 28 29 30 5 31 32 10 14 16 6 33 34 35 36

15. Construct XCOOR, YCOOR and ZCOOR of each subdomain. Since the input data for node coordinates are partitioned and stored among the processors, there are some communications involved in this step. To demonstrate the algorithm in this step, the example in Figure 6.1 is recalled in Figure 6.11. As discussed earlier, the node coordinates of the problem are partitioned and store 6, 5 and 5 node coordinates on processor 0, 1 and 2, respectively.

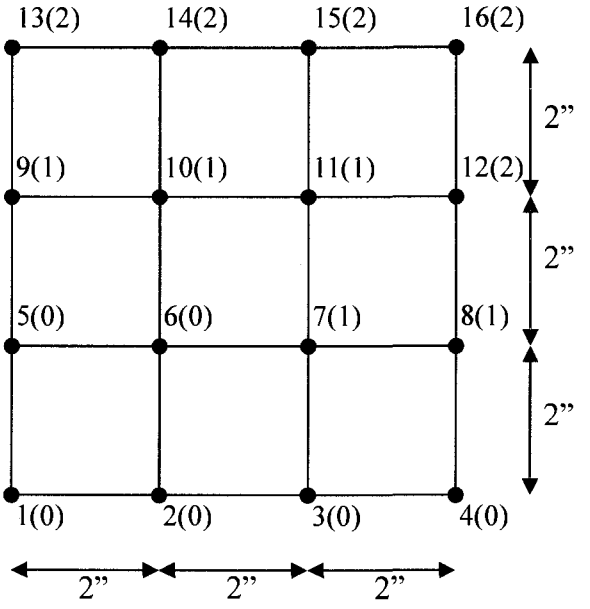


Figure 6.11: A small 4-by-4 example

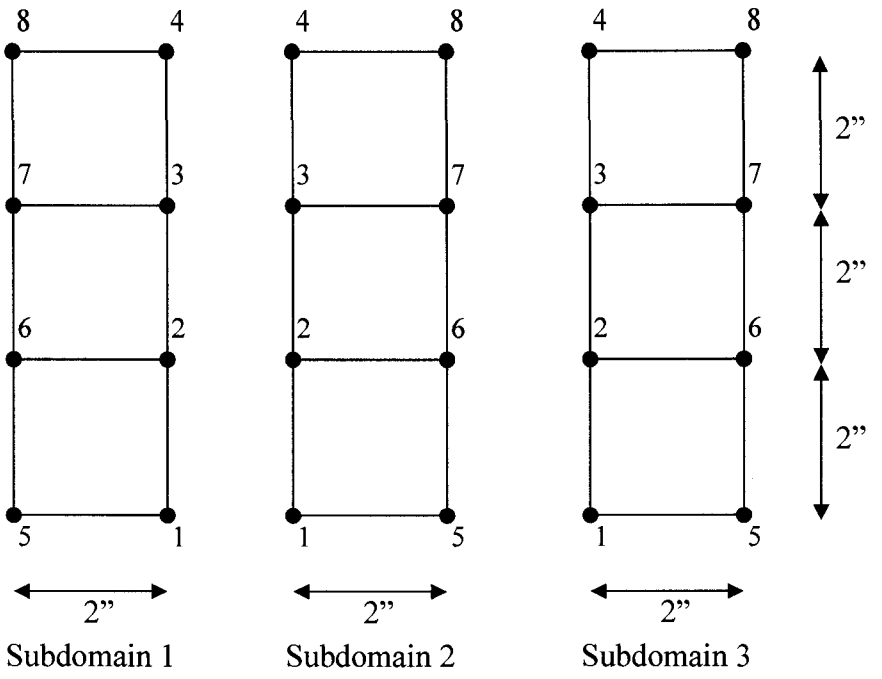


Figure 6.12: Subdomains partitioned from example in Figure 6.11

The domain in this example is assumed to be partitioned as in Figure 6.12. It is clearly seen that node 1-4 of subdomain 1 and 3, node 1-8 of subdomain 2 are boundary nodes.

Below are the steps on how each processor collects node coordinates data from the other processors.

I. Each processor creates list of starting node IDs of each processor. Pseudo code of this step is listed below.

```
Define: ITEMP01(np+1) – Contain the starting node IDs stored on each processor
    where ITEMPO01(i) indicates the starting node ID stored by i^th processor.
ntemp = node/np                    ! initial nodes per cpu
nleft = mod(node,np)               ! left over of node/np
```

```
ITEMP02(1:np) = ntemp          ! store based nodes per cpu to itemp02
ITEMP02(1:nleft) = ntemp+1     ! add the left over to the first nleft processor
ITEMP01(1) = 1
ITEMP01(2:np+1) = ITEMP01(1:np)+ITEMP02(1:np)
```

After the step, each processor has

ITEMP01 = 1 7 12 17

II. The process acquires number of required node information from each processor.

Define:

ITEMP02 of size np+1 contains number of node coordinates of the process owned by (i-1).

ITEMP03 of size nodes contains the list of processes having the coordinate information of local nodes

```
ITEMP02(1:np) = 0
do i = 1,nodes                          ! scan all nodes in the subdomain
    inode is the global node number
    iwho = (inode-1)/ntemp      ! find the guessed process supposed to have i^th
        node information
    do j = iwho,0,-1                    ! this loop is to find the real owner of
        inode
        ilower = ITEMP01(j+1)  ! lower bound of j^th processor
        if inode is greater than or equal to ilower,
            inode belongs to j^th processor.
            ITEMP02(j+1) = ITEMP02(j+1)+1    ! update counter
            ITEMP03(i) = j              ! Record the process actually having node
```

```
                    information
                exit from loop j
        enddo
    enddo
```

After the step, ITEMP02 and ITEMP03 on each processor are;

Processor 0:

ITEMP02 = 4 2 2

ITEMP03 = 0 0 1 2 0 0 1 2

Processor 1:

ITEMP02 = 3 3 2

ITEMP03 = 0 0 1 2 0 1 1 2

Processor 2:

ITEMP02 = 2 3 3

ITEMP03 = 0 1 1 2 0 1 2 2

III. Each processor generates the new form of ITEMP02 and stores in ITEMP04. Basically, ITEMP04(i+1)-ITEMP04(i) is equal to ITEMP02(i), the number of node coordinates of the process owned by $(i-1)^{th}$ processor. Then, each processor has;

Processor 0:

ITEMP04 = 1 5 7 9

Processor 1:

ITEMP04 = 1 4 7 9

Processor 2:

ITEMP04 = 1 3 6 9

IV. ITEMP05 and ITEMP07 arrays are constructed by each processor. In general, ITEMP05 sorts ITEMP03 array such that the nodes that the information stored on the same processor are grouped together. For example, processor 0 groups local nodes 1, 2, 5 and 6 from ITEMP05(1) to ITEMP05(4), local nodes 3 and 7 from ITEMP05(5) to ITEMP05(6) and local nodes 4 and 8 from ITEMP05(7) to ITEMP05(8). Moreover, ITEMP07 array contains the mapping between local nodes and location in ITEMP05 array. The pseudo code in this part is shown below.

```
ITEMP06(1:np+1) = ITEMP04(1:np+1)              ! Temporary array
do i = 1 to nodes
    iown = ITEMP03(i)              ! the owner of the node
    inode = NGBMAP(i)              ! Global node number of node i
    ilocat = ITEMP06(iown+1)       ! location in ITEMP05 to be recorded
    ITEMP06(iown+1) = ITEMP06(iown+1)+1    ! update the location
    ITEMP05(ilocat) = inode        ! record global node ID to ITEMP05
    ITEMP07(ilocat) = i            ! record local node ID to ITEMP07
```

```
enddo
```

Thus, each processor has ITEMP05 and ITEMP07 as;

Processor 0:

ITEMP05 = 2 6 1 5 10 9 14 13

ITEMP07 = 1 2 5 6 3 7 4 8

Processor 1:

ITEMP05 = 2 6 3 10 7 11 14 15

ITEMP07 = 1 2 5 3 6 7 4 8

Processor 2:

ITEMP05 = 3 4 7 11 8 15 12 16

ITEMP07 = 1 5 2 3 6 4 7 8

V. Each Processor uses the information from previous steps to construct node coordinates from its own information. The pseudo code of this part can be expressed below.

```
ist = ITEMP04(me+1)     ! starting location
iend = ITEMP04(me+2)-1                  ! ending location
if ist is not equal to iend;
    do i = ist to iend
        inodeg = itemp05(i)     ! Global node number
        inodel = itemp07(i)     ! Local node number
```

```
ilocat = inodeg-noffmynode      ! local node number stored on the
    processor
xcoor(inodel) = xg(ilocat)                  ! store the global information to
    subdomain information
ycoor(inodel) = yg(ilocat)
if the problem is 3-D problem,
      zcoor(inodel) = zg(ilocat)
enddo
```

Then, each processor extracts node coordinates from its own information and gets partial results as below;

Processor 0:

xcoor = [2.0, 2.0, x, x, 0.0, 0.0, x, x]

ycoor = [0.0, 2.0, x, x, 0.0, 2.0, x, x]

Processor 1:

xcoor = [x, x, 2.0, x, x, 4.0, 4.0, x]

ycoor = [x, x, 4.0, x, x, 2.0, 4.0, x]

Processor 2:

xcoor = [x, x, x, 4.0, x, x, 6.0, 6.0]

ycoor = [x, x, x, 6.0, x, x, 4.0, 4.0]

Note that x indicates the location of the information stored on other

processors.

VI. ITEMP09 array is created on each processor. This step is done in order to find the size of sending data occurring in the next step. From the previous step, ITEMP02 array represents the size of information received from the other processors. On the other hand, ITEMP09 array of size np represents the size of node coordinates sending to each processor. For example, processor 0 sends 3 and 2 node coordinate information to processor 1 and 2, respectively, so ITEMP09 on processor 0 is [4, 3, 2] where the first term indicates the node coordinate information processor 0 has for its own to access. The pseudo code of this part is shown below.

```
ITEMP08(1:np*np) = 0              ! Temporary array
myst = me*np+1                    ! starting location
myend = myst+np-1                         ! ending location
ITEMP08(myst:myend) = ITEMP02(1:np)
All processors call MPI_allreduce to perform MPI_SUM for ITEMP08 array on each
   processors.
ilocat = me+1-np
do i = 1 to np
     ilocat = ilocat+np           ! updating location
     ITEMP09(i) = ITEMP08(ilocat)
enddo
```

ITEMP09 on each processor can be listed below.

Processor 0:

ITEMP09 = 4 3 2

Processor 1:

ITEMP09 = 2 3 3

Processor 2:

ITEMP09 = 2 2 3

VII. All the processors are ready to exchange node coordinate information. Basically, each processor sends the list of nodes that the node coordinate information required to the target processor and receives the node coordinate information back from the target processor. This step requires point to point communication among processors, and the pseudo code can be expressed as below.

```
do I = 1 to np-1
    ipnxt = mod(me+i,np)                ! define sending target
    ipprv = mod(np+me-i,np)             ! define receiving source
    nsend = ITEMP09(ipnxt+1)  ! Number of node coordinates being sent
    nrecv = ITEMP02(ipprv+1)  ! Number of node coordinates being received
    if nrecv is not 0;
        ist = ITEMP04(ipprv+1) ! Starting point of receiving node list
        Send ITEMP05(ist:ist+nrecv-1) to receiving source (ipprv)
    if nsend is not 0;
        Receive sending node list from sending target (ipnxt) and store in
          ITEMP08
        do ii = 1 to nsend
```

```
            il = ITEMP08(ii)-noffmynode          ! Adjusted global node ID
            Store XCOORG(il), YCOORG(il) and ZCOORG(il) to RTEMP01(ii),
              RTEMP01(ii+nsend) and RTEMP01(ii+2*nsend), respectively
        enddo
    if nsend is not 0;
        Receive RTEMP01 array from ipprv processor
        do ii = 1 to the receiving size of RTEMP01
            ioff = ITEMP04(ipprv+1)
            il = ITEMP07(ii+ioff-1)              ! Local node number
            XCOOR(il) = RTEMP01(ii)
            YCOOR(il) = RTEMP01(ii+nrecv)
            ZCOOR(il) = RTEMP01(ii+2*nrecv)
        enddo
enddo
```

Finally, each processor obtains the information of node coordinates as below.

Processor 0:

xcoor = [2.0, 2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0]

ycoor = [0.0, 2.0, 4.0, 6.0, 0.0, 2.0, 4.0, 6.0]

Processor 1:

xcoor = [2.0, 2.0, 2.0, 2.0, 4.0, 4.0, 4.0, 4.0]

ycoor = [0.0, 2.0, 4.0, 6.0, 0.0, 2.0, 4.0, 6.0]

Processor 2:

xcoor = [4.0, 4.0, 4.0, 4.0, 6.0, 6.0, 6.0, 6.0]

ycoor = [0.0, 2.0, 4.0, 6.0, 0.0, 2.0, 4.0, 6.0]

16. Construct external load information of each subdomain. From step 11, the FF array is partitioned in to FB and FI for boundary external load and interior external load, respectively. This part can be done by following the pseudo code below.

```
FB(1:nbdof) = 0.0 and FI(1:nidof) = 0.0
do I = 1 to number of external load acting on the subdomain
    idof = LOADOFS(i)                    ! local dof ID external load applied on
    if idof is greater than nbdof;
        FI(idof-nbdof) = FF(i)
    else;
        FB(idof) = FF(i)
enddo
```

6.4 Efficient way to obtain non-zero locations in K_{bi} and K_{ib} matrices

After each processor obtained IES and JES arrays, subdomain's element connectivity information, the non-zero structure of K_{bi} and K_{ib} matrices could be constructed. Non-zero locations of K_{bi} and K_{ib} matrices are represented in IABI and JABI arrays where JABI array stores the list of interior degrees of freedom associated with local boundary degrees of freedom of the subdomain and IABI(i) indicates the starting location of interior degrees of freedom in JABI array associated with the i^{th} local boundary degree of freedom. The procedures for this phase can be summarized in the pseudo code below.

1. Considering all the elements in the subdomain, including artificial elements from MPC equations, if the element makes a contribution to both boundary and interior nodes, the element will be recorded to MEMKBI array.

```
For each processor:
nmem = 0                ; number of elements associated with both boundary and interior
   nodes
do i = 1 to the number of subdomain's elements
     if the element makes contribution to both boundary nodes and interior nodes
          nmem = nmem+1
          MEMKBI(nmem) = i
     endif
enddo
do i = 1 to the number of subdomain's artificial elements
     if the element makes contribution to both boundary nodes and interior nodes
          nmem = nmem+1
          MEMKBI(nmem) = i
     endif
enddo
```

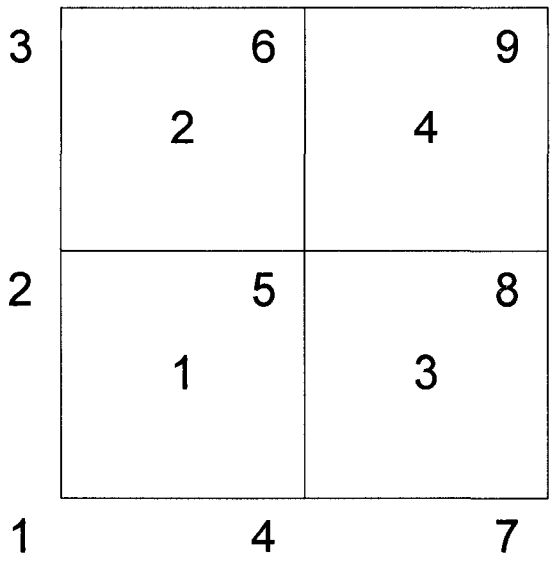


Figure 6.13: A small rectangular element example

To demonstrate the algorithm, a small example in Figure 6.13 is introduced. In this example, there are 4 elements in the subdomain. Each element has 4 nodes, and there are 2 degrees of freedom per node. After step 1, nmem and MEMKBI array of the subdomain are shown below.

nmem = 2

MEMKBI = {1, 2}

2. IABI and JABI arrays, which represent the non-zero locations in K_{bi} , matrix can be obtained from the result from step1. Basically, each boundary node is checked to find the interior nodes attached to the node. The IABI and JABI arrays are constructed. The pseudo code could be expressed below.

```
IABI(1) = 1
```

```
irow = 1                    ; pointer for IABI
ilocale = 0      ; pointer for JABI
do ii = 1 to number of subdomain's boundary nodes
     jicount = 0          ; counter for number of interior nodes associated with ii^th
        boundary node
     do j = 1 to nmem
          if the element consists of ii^th boundary node
               Record the interior node to JETEMP array (avoid recording twice)
               update jicount
          endif
     enddo
     do idof = 1 to number of dofs per node
          irow = irow+1
          IABI(irow) = IABI(irow-1)+jicount*ndofpj
          do im = 1 to jicount
               idofst = starting dof number of the node
               idofend = ending dof number of the node
               do i  = idofst,idofend
                    ilocale = ilocale+1
                    JABI(ilocale) = i
               enddo
          enddo
     enddo
enddo
```

After step 2, IABI and JABI arrays of the subdomain are given below.

IABI = {1, 5, 9, 15, 21, 25, 29}

JABI = {1,2,3,4 ,1,2,3,4 ,1,2,3,4,5,6 ,1,2,3,4,5,6 ,3,4,5,6 ,3,4,5,6}

6.5 Subdomains numerical assembly phase

Once each subdomain obtains element connectivities information of K_{bb} and K_{ii}, the sparse symbolical assembly phase discussed in section 3.3 can be performed in order to acquire non-zero locations of K_{bb} and K_{ii} matrices. As a matter of fact, the non-zero locations of K_{bb} matrix, represented by IABB and JABB arrays, and K_{ii} matrix, represented by IAII and JAII, could be independently formed. However, in the subdomain numerical sparse assembly phase, the numerical values in element stiffness matrices might make contributions to all K_{bb}, K_{bi} and K_{ii} matrices. Therefore, numerical values in K_{bb}, K_{bi} and K_{ii} matrices are constructed during the same phase. An extended version of numerical sparse assembly discussed in (Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications) is introduced to construct the non-zero values of the subdomain's matrices. The pseudo code of this step could be written as below.

```
IP(1:ndofall) = 0
do 40 L = 1 to ndofpe
    Get I = subdomain row dof
    Goto 401 if I is Dirichlet boundary condition
    If i is a boundary dof, assemble diagonal term, ADBB, of K_bb
    If i is a boundary dof, assemble boundary load vector, F_b
    If i is an interior dof, assemble diagonal term, ADII, of K_ii
    If i is an interior dof, assemble interior load vector, F_i
```

set kk = 0 ; indicator that entire row i of $K^{(e)}$ has no contribution to K_{bb} , K_{ii} and K_{bi}

do 20 LL = 1 to ndofpe

Find the location, K, of $K^{(e)}_{\mathrm{L,LL}}$ in the column-wise 1-D array of AE

Find the location, K2, of $K^{(e)}_{\mathrm{LL,L}}$ in the column-wise 1-D array of AE

Goto 20 if L = LL (diagonal term of AE already taken care)

Get J = subdomain column dof

Goto 10 if J is Dirichlet boundary dof

Goto 20 if J is less than I ; skip the lower part, it is already taken care using K2

IP(J) = K ; record Jth column, which associates with Ith row & the correspond Kth location of AE (i.e. $K^{(e)}_{\mathrm{L,LL}}$)

IP2(J) = K2 ; record Jth column, which associates with Ith row & the corresponding K2th location of AE (i.e. $K^{(e)}_{\mathrm{L,LL}}$)

set kk = 1 ; indicator that row L of $K^{(e)}$ makes contribution to K_{bb} , K_{ii} and K_{bi}

goto 20

10 continue

if I and J are boundary dofs, $F_b(i) = F_b(i) - F_b(j) \cdot AE(k)$

if I is boundary and J is interior, $F_b(i) = F_b(i) - F_i(j - nbdof) \cdot AE(k)$

if I is interior and J is boundary, $F_i(i - nbdof) = F_i(i - nbdof) - F_b(j) \cdot AE(k)$

if I and J are interior dofs, $F_i(i - nbdof) = F_i(i - nbdof) - F_i(j - nbdof) \cdot AE(k)$

20 continue

if KK is 0 goto 40 ; row L of $K^{(e)}$ makes no contribution

if I is a boundary dof,

do 30 J = IABB(I) to IABB(I+1)-1

icol = JABB(J)

K = IP(icol)

K2 = IP2(icol)

if K is 0 goto 30 ; the current element has nothing to do with K_{bb} , K_{ii} and K_{bi}

ANBB(J) = ANBB(J)+AE(K)

if K2 is 0 goto 30

ANBB2(J) = ANBB2(J)+AE(K2)

IP(icol) = 0 and IP2(icol) = 0 ; reset the value before considering the next row

30 continue

do 31 J = IABI(I) to IABI(I+1)-1

icol = JABI(J)+nbdof

K = IP(icol)

K2 = IP2(icol)

if K is 0 goto 31 ; the current element has nothing to do with K_{bb} , K_{ii} and K_{bi}

ANBI(J) = ANBI(J)+AE(K)

if K2 is 0 goto 31

ANBI2(J) = ANBI2(J)+AE(K2)

IP(icol) = 0 and IP2(icol) = 0 ; reset the value before considering the next row

31 continue

if I is an interior dof,

do 32 J = IAII(I-nbdof) to IAII(I-nbdof+1)-1

icol = JAII(J)+nbdof

K = IP(icol)

K2 = IP2(icol)

if K is 0 goto 32 ; the current element has nothing to do with K_{bb} , K_{ii} and K_{bi}

ANII(J) = ANII(J)+AE(K)

if K2 is 0 goto 32

ANII2(J) = ANII2(J)+AE(K2)

IP(icol) = 0 and IP2(icol) = 0 ; reset the value before considering the next row

32 continue

goto 40

401 continue

if I is a boundary dof, ADBB(I) = 1.0

if I is an interior dof, ADII(I-nbdof) = 1.0

40 continue

CHAPTER VII

CONCLUSION AND FUTURE RESEARCH

7.1 Conclusion

MPI/FORTRAN finite element analysis software based on Domain Decomposition formulation has been developed. Efficient input data storage/data communication schemes, domain partitioning, fast symbolical and numerical sparse assembly, symmetrical/unsymmetrical sparse solver and robust symmetrical/unsymmetrical iterative solvers algorithms are all utilized in the developed code. The code has been developed in MPI/FORTRAN and can effortlessly be ported to other computer platforms (Watson, Nark and Nguyen). Moreover, the use of a distributed data storage scheme for the input data, domain partitioning and symmetrical iterative solver can benefit users by solving large-scale problems on distributed memory computers.

The developed code in this work is working as stand-alone finite element analysis software where users provide problem information, such as number of equations, element connectivity, node coordinates, load and boundary conditions. Before performing the analysis, ParMETIS is performed so as to find the subdomain in which each node belongs, and the results from ParMETIS requires further computation since the nodes in the domain need to be distinguished as boundary nodes and/or interior nodes. Each processor then obtains its own subdomain's information, such as; element connectivity, node coordinates, boundary conditions, load conditions and material properties. After that, subdomain coefficient matrices related to boundary and interior degrees of freedom are constructed, and factorization of subdomain's interior degrees of freedom coefficient

matrix is performed. Due to limitation of computer memory available on distributed memory computers, Preconditioned Conjugate Gradient (PCG) and Flexible Generalized Minimum Residual (FGMRES) are chosen as symmetrical and unsymmetrical iterative solvers, respectively (for solving system's nodal boundary dofs). Lastly, subdomain's interior dofs for each subdomain are computed by direct sparse solvers, and global solution vector is constructed as the output.

The performance of two acoustic examples with various numbers of grids is conducted to demonstrate the accuracy and efficiency of the developed code. The first example is a 3-D "symmetrical" acoustic application, and the second example is a 2-D "unsymmetrical" acoustic application. The results obtained from ODU Wilbur (parallel) cluster have revealed the super-linear speedup in 3-D symmetrical acoustic example. In addition, the robustness (and efficiency) of the developed code has been observed in both symmetrical and unsymmetrical examples. Regarding the computer in-core memory usage, the developed code has shown its ability to efficiently solve large-scale problems on distributed memory machines.

7.2 Future research

According to the dissertation work discussed herein, the following future researches are suggested.

1. Investigate the possibility of further time reduction in calculating the triple product in equation 2.12, by employing the "partial" LDL transpose derivations suggested by (Komzsik).

2. Develop a stand-alone (none finite element based) DD equation solver.

3. Develop a parallel direct ("not" mixed direct-iterative) solver to solve for system's boundary displacements.

REFERENCES

Amestoy, P. R., I. S. Duff and J. Y. L'Excellent. Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers. Technical Report RAL-98/51. Oxon, England: Rutherford Appleton Laboratory, 1998.

—. Mumps Multifrontal Massively Parallel Solver Version 2.0. Technical Report TR/PA/98/02. Toulouse, France: CERFACS, 1998.

Chen, K. Matrix Preconditioning Techniques and Applications. Cambridge, UK: Cambridge UP, 2005.

Dongarra, J. J., et al. Numerical Linear Algebra for High-Performance Computers. Philadelphia: SIAM, 1998.

Farhat, C., and F. X. Roux. "Implicit Parallel Processing in Structural Mechanics." Computational Mechanics Advances (1994): 1-124.

Farhat, C., M. Lesoinne and P. LeTallec. "FETI-DP: A Dual-Primal Unified FETI Method - Part I: A Faster Alternative to the Two-Level FETI Method." International Journal for Numerical Methods in Engineering (2001): 1523-1544.

Hestenes, M. R., and E. Stiefel. "Methods of Conjugate Gradients for Solving Linear Systems." Journal of Research of the National Bureau of Standards (1952): 409-436.

Karypis, G., K. Schloegel and V. Kumar. ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Minneapolis, 15 August 2003.

Komzsik, Louis. What Every Engineer Should Know About Computational Techniques of Finite Element Analysis. Boca Raton, Fl: Taylor & Francis, 2005.

MPI: A Message-Passing Interface Standard. 12 June 1995. 19 March 2008

<http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>.

MPI2: Extensions to the Message-Passing Interface. 18 July 1997. 19 March 2008 <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>.

Nguyen, D. T. Finite Element Methods: Parallel Sparse Statics and Eigen-Solutions. New York: Springer, 2006.

—. Parallel-Vector Equation Solvers for Finite Element Engineering Applications. New York: Kluwer/Plenum, 2002.

Nguyen, D. T., et al. "Parallel Finite Element Domain Decomposition for Structural/Acoustic Analysis." Journal of Computational and Applied Mechanics 4(2): 2003. 189-201.

Przemieniecki, J. S. "Matrix Structural Analysis of Substructures." AIAA (1963): 138-147.

Rajan, S. D. "Introduction to Structural Analysis & Design." S.D. Rajan, ed. Introduction to Structural Analysis & Design. New York: Wiley, 2000. 419-421.

Razzaq, Zia, et al. "Substructure Analysis Using NICE SPAR and Application of Force to Linear and Nonlinear Structures." NASA Technical Report (NAS 1.26180317; NASA-CR-180317). Aug 1, 1987.

Saad, Y. "A Flexible Inner-Outer Preconditioned GMRES Algorithm." SIAM Journal on Scientific and Statistical Computing (1993): 461-469.

—. "Yousef Saad -- Books." 3 January 2000. 29 January 2008 <http://www-users.cs.umn.edu/~saad/books.html>.

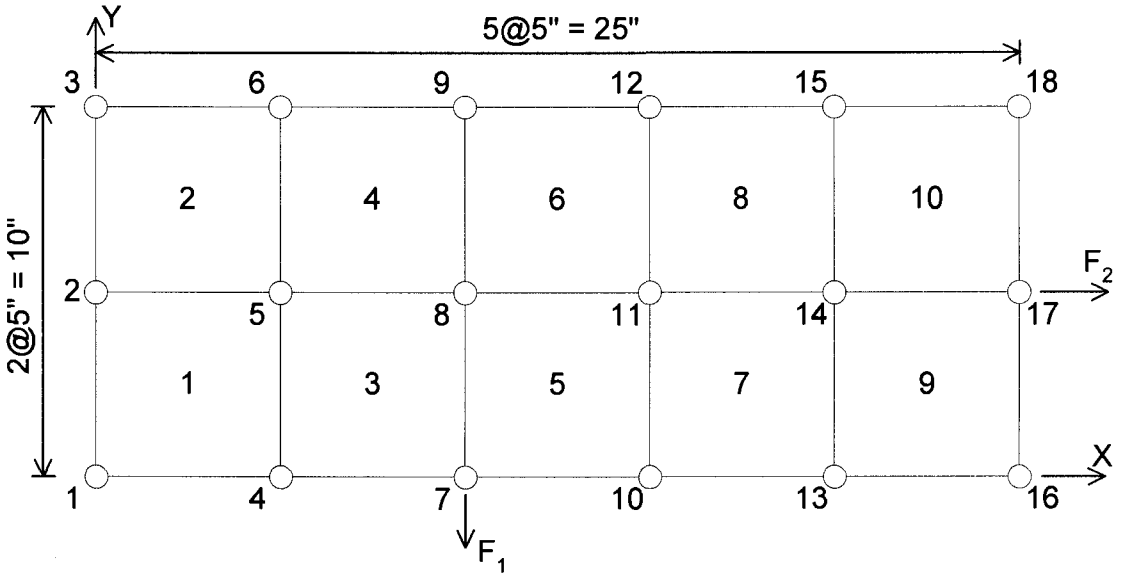
TOP500.Org. "TOP500 List - November 2007." November 2007. TOP500 Supercomputing Sites. 20 March 2008 <http://www.top500.org/list/2007/11/100>.

Watson, W. R. "Three-Dimensional Rectangular Duct Code With Application to Impedance Eduction." AIAA Journal (2002): 217-226.

Watson, W. R., et al. "On the Development of an Efficient Parallel Hybrid Solver with Application to Acoustically Treated Aero-Engine Nacelles." NASA Technical Report. Dec 06, 2006.

APPENDIX A

A.1 INPUT AND OUTPUT DATA FORMAT FOR CDDFEA SUBROUTINE



Number of dof per node : 2
Number of CPU : 3
Dirichlet boundary dof @ dof 1, 3 and 5
Multi point constraint equations are as below.

$x_{10}-2x_{23} = 0$

$x_1+2x_{21}-5x_{35} = -2$

Figure A1.1: 18 node, 10 rectangular element example

The example in Figure A1.1 will be used to illustrate the input and output format of the code.

1. Element connectivity information [IE(nsizeiea), JE(nsizejea)]

Element connectivity information is stored in IE and JE arrays. The structure of the arrays is represented by the distributed compressed storage row format

(distributed CSR) explained in chapter 6.1. Basically, distributed CSR is the extended storage scheme of CSR format. We will describe the IE and JE arrays in CSR format, and then describe how to store the arrays in distributed CSR format among processors.

Serial CSR format:

The size of IE, an integer array, is [number of element+1 , or nel+1] and the size of JE, also an integer array, is [IE(nel+1)-1]. Therefore, for the particular example, the size of IE is 11 (nel+1=10+1), and the size of JE is 40. IE and JE of the problem can be shown below.

$$IE = [1,5,9,13,17,21,25,29,33,37,41]^T$$

$$JE = [\ 1\ \ 2\ \ 5\ \ 4\ \ 2\ \ 3\ \ 6\ \ 5\ \ 4\ \ 5$$
$$8\ \ 7\ \ 5\ \ 6\ \ 9\ \ 8\ \ 7\ \ 8\ \ 11\ \ 10$$
$$8\ \ 9\ \ 12\ \ 11\ \ 10\ \ 11\ \ 14\ \ 13\ \ 11\ \ 12$$
$$15\ \ 14\ \ 13\ \ 14\ \ 17\ \ 16\ \ 14\ \ 15\ \ 18\ \ 17\ \ ]^T$$

Distributed CSR format:

This storage scheme is an extension of the CSR format. The idea of this scheme is to distribute the CSR format arrays among processors. The advantage of this scheme over serial CSR format is that bigger problem sizes can be solved on distributed memory machines since each processor will store just a portion of the connectivity arrays. For this particular example, the size of IE arrays on

processor 0, 1 and 2 are 5, 4 and 4, respectively. Also, the size of JE arrays on processor 0, 1 and 2 are 16, 12 and 12, respectively. IE and JE of the problem in distributed CSR format are shown below.

Processor 0:

$$IE = [1,5,9,13,17]^T$$

$$JE = [1,2,5,42,3,6,5,4,5,8,7,5,6,9,8]^T$$

Processor 1:

$$IE = [1,5,9,13]^T$$

$$JE = [7,8,11,10,8,9,12,11,10,11,14,13]^T$$

Processor 2:

$$IE = [1,5,9,13]^T$$

$$JE = [11,12,15,14,13,14,17,16,14,15,18,17]^T$$

2. Dirichlet boundary conditions information [nbc, NBCDOF(nbc)]

The dirichlet boundary conditions information is represented by nbc, scalar number, and NBCDOF array. nbc is an integer number indicating the number of dirichlet boundary conditions of the domain, which is 3 in this example. NBCDOF is an integer array containing the list of dirichlet boundary condition dof of the domain. For this particular example, nbc is 3 and NBCDOF is $[1\ 2\ 3]^T$.

3. Joint coordinates information [XCOORG(nsizemynode), YCOORG(nsizemynode), ZCOORG(nsizemynode)]

XCOORG, YCOORG and ZCOORG are double precision arrays whose sizes are nsizemynode as discussed in chapter 6.1. As a result, for the example in Figure A1.1, nsizemynode of processor 0, 1 and 2 are 6, 6 and 6, respectively. XCOORG, YCOORG and ZCOORG on all processors can be expressed as below.

Processor 0:

$$XCOORG = [0.0,0.0,0.0,5.0,5.0,5.0]^T$$

$$YCOORG = [0.0,5.0,10.0,0.0,5.0,10.0]^T$$

Processor 1:

$$XCOORG = [10.0,10.0,10.0,15.0,15.0,15.0]^T$$

$$YCOORG = [0.0,5.0,10.0,0.0,5.0,10.0]^T$$

Processor 2:

$$XCOORG = [20.0,20.0,20.0,25.0,25.0,25.0]^T$$

$$YCOORG = [0.0,5.0,10.0,0.0,5.0,10.0]^T$$

Note: Since the example is a 2D problem, ZCOORG is not applicable in this case.

4. External load information [nloadofg, LOADOFSG(nloadofg), FFG(nloadofg)]

External load information is stored in nloadofg, LOADOFSG, FFG. nloadofg is an integer number indicating the number of external loads acting on the domain. LOADOFSG is an integer array storing the list of degrees of freedom attached by the external loads. The size of LOADOFSG is the number of external loads, nloadofg. FFG is a complex array storing the value of the external loads at each degree of freedom. In fact, the size of FFG is the same as LOADOFSG, which is nloadofg. For this example, nloadofg, LOADOFSG and FFG can be expressed as below.

$$nloadofg = 2$$

$$LOADOFSG = [14,33]^T$$

$$FFG = [-F_1, F_2]^T$$

5. Material properties information [npropmat, MEMATER(nel), CPROP(200), IPROP(200), RPROP(200)]

Material properties information is stored in npropmat, MEMATER, CPROP, IPROP and RPROP. npropmat is the number of material sets in the domain. MEMATER is an integer array containing the material set id of each element. The size of MEMATER is the number of elements, nel. CPROP is a complex array containing the material properties values of all material sets. IPROP is a long integer array containing the material properties values of all material sets. Also, RPROP is a double precision array containing the material properties values of all material sets. The size of CPROP, IPROP and RPROP are set to be 200. Elements

in CPROP, IPROP and RPROP can be described in the Table below.

Location in CPROP, IPROP and RPROP	Description
1-10	Reserved for material properties of element type 1
11-20	Reserved for material properties of element type 2
21-30	Reserved for material properties of element type 3
31-40	Reserved for material properties of element type 4 (3D symmetrical acoustic element)
41-50	Reserved for material properties of element type 5 (3D symmetrical acoustic element)
51-60	Reserved for material properties of element type 6 (2D symmetrical and unsymmetrical acoustic element)

6. Multi-point constraint equations information [nmpcg, IAMPCG(nmpcg+1), JAMPCG(IAMPCG(nmpcg+1)-1), CMPCG(IAMPCG(nmpcg+1)-1), RMPCG(nmpcg)]

From the MPC equations discussed in chapter 4.5, the information of MPC equations could be stored in variables as below.

$$nmpcg = 2$$

$$IAMPCG = [1,3,6]^T$$

$$JAMPCG = [10,23,1,21,35]^T$$

$$CMPCG = [1.0,-2.0,1.0,-2.0,-5.0]^T$$

$$RMPCG = [0.0,-2.0]^T$$

7. Miscellaneous information [NCUM(neltype+1), IDDPAR(200)]

NCUM array is an integer array containing the information of number of elements in each element type. Basically, number of elements of i^{th} element type will be ncum(i+1)-ncum(i). The size of NCUM array is the number of element type plus 1.

IDDPAR is a long integer array containing the program information. The size of IDDPAR is 200 and the details of each element will be explained below.

iddpar(1) – me, Processor id

iddpar(2) – np, Number of processors

iddpar(3) – memavai, Memory available for each processor (unit: bytes)

iddpar(4) – reserved

iddpar(5) – reserved

iddpar(6) – reserved

iddpar(7) – reserved

iddpar(8) – nel, Number of elements

iddpar(9) – nbc, Number of Dirichlet boundary conditions

iddpar(10) – node, Number of nodes

iddpar(11) – ndofpn, Number of degrees of freedom per node

iddpar(12) – npe, Number of nodes per element

iddpar(13) – ndofpe, Number of degrees of freedom per element

iddpar(14) – nloadofg, Number of external loads

iddpar(15) – nmatprop, Number of material properties sets

iddpar(16) – iprob, Problem type (1: 3D Acoustic problem, 2: 2D Acoustic problem, 3: 3D Acoustic problem having properly required input data format)

iddpar(17) – iway, Domain breaking scheme (0: ParMETIS, 1: Author's scheme)

iddpar(18) – iter, Type of the solver used to solve for boundary dofs (0: direct solver, 1: PCG (symmetrical problem) and Bicg-stab(unsymmetrical problem), 2: GMRES (unsymmetrical problem), 3: FGMRES (unsymmetrical problem).

iddpar(19) – ireord, Reordering scheme to minimizing fill-ins (0: no reordering, 1:METIS)

iddpar(20) – islvr, =1 (reserved for future extension)

iddpar(21) – neltype, Number of supported element types (= 6 for current package)

iddpar(22) – isy, =0 (reserved for future extension)

iddpar(23) – ncoef, Acoustic parameter

iddpar(24) – neq, Number of equations

iddpar(25) – ibgen, Preconditioned scheme used in iterative solver (0: no preconditioning, 1: obtained preconditioned matrix from K_{bb}, 2: obtained preconditioned matrix from $\overline{K}_{bb}$ assuming that $K_{ii}^{(r)}$ is a diagonal matrix.

iddpar(26) – iunr, =1 (reserved for future extension)

iddpar(27) – ierrchk, Error checking flag (0: no error checking, 1: with error checking)

iddpar(31) – reserved

iddpar(32) – reserved

iddpar(33) – reserved

iddpar(34) – reserved

iddpar(35) – ipartieje, element connectivity partitioning scheme (0: no partitioning of ie and je, 1: ie and je will be partitioned and distributed among CPUs)

iddpar(36) – nsizeiea, Size of ie after partitioning

iddpar(37) – nsizejea, Size of je after partitioning

iddpar(38) – noffiejea, Offset of ie after partitioning (i.e. the number of elements owned by the processors having ID less than me)

iddpar(39) – memused, Total memory used before calling cddfea subroutine

iddpar(40) – istop, This parameter is used for debugging purpose

iddpar(41) – ilast, This parameter is used for debugging purpose

iddpar(42) – nsizemynode, number of nodes stored on the processor

iddpar(43) – noffmynode, Offset of the node ID (i.e. the number of nodes owned by the processors having ID less than me)

iddpar(44) – i3d, 3-dimensional flag (1: 3 dimensional problem, 0: 2 dimensional problem)

Upon the completion of the code, ifin is 1, the code will return a complex array, XSOL(neq), as the solution output.

A.2 INSTRUCTIONS FOR USERS TO ADD A NEW FINITE ELEMENT TYPE INTO THE PACKAGE

Before going into the details of the instruction, it should be noted that there are some limitations of the current version of the code as described below.

1. The maximum number of nodes per element is set to be 8.
2. Only 1 element type in a problem has been fully tested. In the future, any problem can have as many element types as needed.
3. Only 1 material set in a problem has been fully tested. In the future, any problem can have as many material sets as needed.

It should be noted that users should follow the instruction in Appendix A.1 for the proper format of the input data.

To add a new finite element into the code, the following steps need to be followed:

1. Define the ID of the new finite element. There are 6 element type slots in the code, and the 5^{th} and 6^{th} slots are respectively occupied by 3-D and 2-D acoustic finite elements. In addition, the 4^{th} type is used to demonstrate the steps to add 3-D acoustic finite element into the code. Therefore, users can use element type 4 as an example.
2. There are a few places users have to modify in NUMASS subroutine. The pseudo FORTRAN code shown in Table A2.1 is used to demonstrate the flow in NUMASS subroutine, which is the subroutine to construct the coefficient matrix and load vector of the problem.

Table A2.1: Pseudo FORTRAN code of NUMASS subroutine

```
      do 2 ii=1,neltype
           nmems=ncums(ii+1)-ncums(ii)
           if(nmems.eq.0) go to 2
           jstart=jend+1
           jend=jstart+nmems-1
           go to (11,12,13,14,15,16),ii
      11    continue
           go to 2
      12    continue
           go to 2
      13    continue
           go to 2
      14    continue              ! General Problem
           ndofpe = npe*ndofpn
c     STEP 4.1 : Each processor generates problem parameters.
           ns = nnx*nny*ndofpn
           nnx  = iprop(41)
           nny  = iprop(42)
           nnz  = iprop(43)
           ifreq = iprop(44)
           allocate(ctemp01(nnx*nny+1))
           call acousticsym(ndofpe,ifreq,nnx,nny,nnz,ctemp01)
c     STEP 4.2 : Each processor begins elements loop.
           do 40 ie = jstart,jend
c     STEP 4.3 : Each processor finds local node ids before reordering associated
      with ieth element
```

```
                if(npe.ge.1) lm(1) = node1(ie)
                if(npe.ge.2) lm(2) = node2(ie)
                if(npe.ge.3) lm(3) = node3(ie)
                if(npe.ge.4) lm(4) = node4(ie)
                if(npe.ge.5) lm(5) = node5(ie)
                if(npe.ge.6) lm(6) = node6(ie)
                if(npe.ge.7) lm(7) = node7(ie)
                if(npe.ge.8) lm(8) = node8(ie)
c   STEP 4.4 : Reset Element stiffness matrix and load vector.
                elk(1:ndofpe**2) = 0.
                be(1:ndofpe) = 0.
c   STEP 4.5 : Compute information required to construct element stiffness
    matrix and load vector.
                nodeid = lm(1)
                xx = xcoor(nodeid)
                yy = ycoor(nodeid)
                zz = zcoor(nodeid)
                call elemlocat(xx,yy,zz,ix,iy,iz)
c   STEP 4.6 : Call subroutine to compute element stiffness matrix and load
    vector
                call linear1(ix,iy,iz,nnx,nnz,ns,elk,be,ctemp01)
c   STEP 4.7 : Reorder lm array
                call lmreord(ndofpe,lm,juii)
c   STEP 4.8 : Call Numerical assembly subroutine (sym or unsym)
                if (isy .eq. 1) then
                     call numassdd(nbj,ndofpn,ndofpe,lm,idir,elk,be,iabb,jabb
                $     ,ip,iabi,jabi,iaii,jaii,adbb,bb,adii,bi,anbb,anbi,anii
                $     ,iperm,invp,me)
```

```
                else
                    call numassddunsym(nbj,ndofpn,ndofpe,lm,idir,elk,be,iabb
                $     ,jabb,ip,iabi,jabi,iaii,jaii,adbb,bb,adii,bi,anbb,anbi,anii
                $     ,iperm,invp,me,itempo1,anbb2,anbi2,anii2,elk)
                endif
c   STEP 4.9 : The end of loop 40
          40    continue
c   *** This part is for 3D acoustic problem only
c   *** Users have to remove do 45 loop for other problem types
c   impose boundary condition to the system
          do 45 i = 1,ndir
              ib = nbcdofs(i)              ! Dirichlet bdof
              ibnew = juii(ib)             ! reordered Dbdof
              if (ibnew .gt. nbdof) then     ! interior dof
                  ibnew2 = ibnew-nbdof       !
                  bi(ibnew2) = ctemp01(i)
              else
                  bb(ibnew) = ctemp01(i)
              endif
          45  continue
c   STEP 4.10 : Deallocate all working arrays
          deallocate(ctemp01)
          goto 2
       15   continue
          goto 2
       16   continue
          goto 2
       2    continue
```

3. In NUMASS subroutine, loop 2 is the element type loop, which means that all the element types in the subdomain are included in this step to calculate element stiffness matrices and load. Based on the new finite element ID the user set in the input data, the code will skip to appropriate place in loop 2. In other words, element type ID 1, 2 3, and 4 will go to label 11, 12, 13 and 14, respectively. To simplify the discussion, element type 4 is used to demonstrate the procedure of this section.

4. After label 14, users have to add the following items.

   4.1. Each Processor generates problem parameters based on input CPROP, IPROP and RPROP arrays. If temporary arrays are required in order to construct the element stiffness matrix in this step, the user can allocate

   CTEMP01, CTEMP02, CTEMP03 and CTEMP04 for double complex arrays;

   ITEMP01, ITEMP02, ITEMP03 and ITEMP04 for integer arrays;

   I8TEMP01, I8TEMP02, I8TEMP03 and I8TEMP04 for long integer arrays;

   RTEMP01, RTEMP02, RTEMP03 and RTEMP04 for double precision arrays.

   4.2. Each processor begins element loop from jstart to jend. The value of jstart and jend are already defined by the code, so the user does not have to change these values.

   4.3. Inside the element loop, the local node numbers before reordering of ie^{th} element are obtained. Users should use the code illustrate in the example

without modification in this part.

4.4. Each processor initializes element stiffness matrix, ELK, and element load vector, BE.

4.5. Each processor computes parameters required to construct the element stiffness matrix and element load vector.

4.6. Element stiffness subroutine is called in order to obtain the element stiffness matrix and element load vector. The element stiffness matrix is stored row-wised in ELK array, which is a 1 dimensional array, and the element load vector is stored in BE array.

4.7. LMREORD subroutine is called to reorder the local node number. Users can use this part of the code in the example provided.

4.8. NUMASSDD subroutine is called for symmetrical problem, and NUMASSDDUNSYM subroutine is called for unsymmetrical problem. Again, users can use this part of the code in the example provided.

4.9. Each processor performs step 4.2 again until all elements in the element type are included in the computation.

4.10. Each processor deallocates all working arrays in this part.

APPENDIX B

B.1 FLOWCHART OF THE DEVELOPED CODE

Once the main subroutine of the developed code is called (see Appendix A.1 for Input data format for the code), the following steps are performed. For the variable descriptions, please see Appendix B.2 for details.

1. Domain partitioning (CDDBREAK subroutine)

   Description: The main subroutine to break the entire domain into subdomains. In this subroutine, ParMETIS is called in order to find the owner of each node. Then, further computations are performed to obtain subdomain's information.

   Argument list: me, np, iway, IE, JE, iprob, nbc, NBCDOF, NCUM, ndofpe, nel, neltype, node, npe, ndofpn, nloadofg, LOADOFSG, FFG, NODE1, NODE2, NODE3, NODE4, NODE5, NODE6, NODE7, NODE8, NCUMS, nels, nbj, nbjall, nbdof, nidof, ndofall, ndofalls, IIX, IIY, IIZ, IBCB, IBCI, IBDOFARE, nloadof, LOADOFS, FF, BB, BI, NBJGLOB, ngbjdof, NGBMAP, nodes, nsub, ndir, NBCDOFS, IRITE, io3, io7, nmpcg, IAMPCG, JAMPCG, CMPCG, RMPCG, nmpc, IAMPCC, JAMPC, CMPC, RMPC, iflag, ipartieje, nsizeiea, nsizejea, noffiejea, ifin, ncoef, nnx, nny, nnz, ma, na

   Input

me	np
iway	IE(nsizeiea)
JE(nsizejea)	iprob
nbc	NBCDOF(nbc)
NCUM(neltype+1)	ndofpe
nel	neltype
node	npe
ndofpn	nloadofg
LOADOFSG(nloadofg)	FFG(nloadofg)

ndofall	nsub
IRITE(20)	io3
io7	nmpcg
IAMPCG(nmpcg+1)	JAMPCG(IAMPCG(nmpcg+1)-1)
CMPCG(IAMPCG(nmpcg+1)-1)	RMPCG(nmpcg)
iflag	ipartieje
nsizeiea	nsizejea
noffiejea	ifin
ncoef	nnx
nny	nnz
ma	na

Output:

NODE1(nels)	NODE2(nels)
NODE3(nels)	NODE4(nels)
NODE5(nels)	NODE6(nels)
NODE7(nels)	NODE8(nels)
NCUMS(neltype+1)	nels
nbj	nbjall
nbdof	nidof
ndofalls	IIX(nels)
IIY(nels)	IIZ(nels)
IBCB(nbdof)	IBCI(nidof)
IBDOFARE(nbdof)	nloadof
LOADOFS(nloadof)	FF(nloadof)**
BB(nbdof)	BI(nidof)
NBJGLOB(nodes)	ngbjdof
NGBMAP(nodes)	nodes
ndir	NBCDOFS(ndir)
nmpc	IAMPC(nmpc+1)
JAMPC(IAMPC(nmpc+1)-1)	CMPC(IAMPC(nmpc+1)-1)
RMPC(nmpc)	ifin

2. Subdomain's element connectivities (DDCONNECT subroutine)

Description: Each processor constructs element connectivities of its own subdomain in this phase.

Argument list: me, np, neltype, mndofpj, nbj, nbdof, NODE1, NODE2, NODE3, NODE4, NODE5, NODE6, NODE7, NODE8, NCUMS, NKBITYPE, npe, IEBB, IEII, IABI, LM, JEBB, JEII, JABI, JETEMP, MEMNAKBI, nmpc, iampc, jampc

Input:

me	np
neltype	mndofpj
nbj	nbdof
NODE1(nels)	NODE2(nels)

NODE3(nels)
NODE4(nels)
NODE5(nels)
NODE6(nels)
NODE7(nels)
NODE8(nels)
NCUMS(neltype+1)
NKBITYPE(neltype+2, temporary)
npe
LM(npe+100, temporary)
JETEMP(nbdof, temporary)
MEMNAKBI(nels, temporary)
nmpc
IAMPC(nmpc+1)
JAMPC(IAMPC(nmpc+1)-1)

Output:

IEBB(nels+nmpc+1)
IEII(nels+nmpc+1)
IABI(nbdof+1)
JEBB(JEBB(nels+nmpc+1)-1)
JEII(IEII(nels+nmpc+1)-1)
JABI(IABI(nbdof+1)-1)

3. Reordering of the subdomain coefficient matrix (REORD subroutine)

Description: From adjacency information, a reordering scheme is performed in order to reduce fill-in terms during the factorization phase

Argument list: n, ireord, IAKEEP, JA, IPERM, INVP, IT

Input:

nidof
ireorder
IAKEEP(n+1)
JA(IAKEEP(n+1)-1)
IT(5*ncoef1, temporary)

Output:

IPERM(n)
INVP(n)

4. Symbolic assembly of subdomain's coefficient matrices (SYMBASSREORD subroutine)

Description: Symbolic assembly phase to obtain non-zero information of K_{bb}, K_{bi}, K_{ii}

Argument list: IA, JA, IAKEEP, JAKEEP†, IPERM, INVP, n, ncoef1

Input:

IAKEEP(n+1)
JAKEEP†(IAKEEP(n+1)-1)
IPERM(n)
INVP(n)
n

Output:

IA(n+1)	JA(IA(n+1)-1)

† This JAKEEP is the same as JA array from reordering phase.

5. Numerical assembly of subdomain's coefficient matrices (NUMASS subroutine)

Description: Numerical assembly phase to obtain non-zero information of K_{bb}, K_{bi}, K_{ii}

Argument list: ndir, nbcdofs, neltype, NCUMS, ndofpn, nbj, IP, IDIR, npe, NODE1, NODE2, NODE3, NODE4, NODE5, NODE6, NODE7, NODE8, LM, ELK, BE, IABB, JABB, IABI, JABI, IAII, JAII, BI, ANBB, ANBI, ANII, ADBB, ADII, BB, ITEMP01, MEMATER, XCOOR, YCOOR, ZCOOR, IPROP, CPROP, RPROP, numater, nsub, me, ANBB2, ANBI2, ANII2, NGBMAP, LB, ELK2, ITEMP02, ielem, isy, ndofalls, nbdof, nidof, nmpc, IAMPC, JAMPC, CMPC, RMPC, IIX, IIY, IIZ, nnx, nny, nnz, IPERM, INVP, ma, na, xmach, wn, wy

Input:

ndir	nbcdofs
neltype	NCUMS(neltype+1)
ndofpn	nbj
IP(ndofalls, temporary)	IDIR(ndofalls, temporary)
npe	NODE1(nels)
NODE2(nels)	NODE3(nels)
NODE4(nels)	NODE5(nels)
NODE6(nels)	NODE7(nels)
NODE8(nels)	LM(ndofpe, temporary)
ELK(ndofpe^2, temporary)	BE(ndofpe, temporary)
IABI(nbdof+1)	JABI(IABI(nbdof+1)-1)
ITEMP01(ndofalls, temporary)	MEMATER(nels)
XCOOR(nodes)	YCOOR(nodes)
ZCOOR(nodes)	IPROP(200)
CPROP(200)	RPROP(200)
numater	nsub
me	NGBMAP(nodes)
LB(ndofpe, temporary)	ELK2(ndofpe^2, temporary)
ITEMP02(ndofalls, temporary)	ielem
isy	ndofalls

nbdof	nidof
nmpc	IAMPC(nmpc+1)
JAMPC(IAMPC(nmpc+1)-1)	CMPC(IAMPC(nmpc+1)-1)
RMPC(nmpc)	IIX(nels)
IIY(nels)	IIZ(nels)
nnx	nny
nnz	IPERM(nidof)
INVP(nidof)	ma
na	xmach
wn	wy

Output:

IABB(nbdof+1)	JABB(IABB(nbdof+1)-1)
IAII(nidof+1)	JAII(IAII(nidof+1)-1)
BI(nidof)	ANBB(IABB(nbdof+1)-1)
ANBI(IABI(nbdof+1)-1)	ANII(IAII(nidof+1)-1)
ADBB(nbdof)	ADII(nidof)
BB(nbdof)	ANBB2(IABB(nbdof+1)-1)
ANBI2(IABI(nbdof+1)-1)	ANII2(IAII(nidof+1)-1)

6. Symbolic factorization of Kii (SYMFACTCHK subroutine)

Description: Symbolic factorization phase for K_{ii}

Argument list: nidof, IA, JA, IU, JU, IP, ncoef2, npred, iflag

Input:

nidof	IA(nidof+1)
JA(IA(nidof+1)-1)	IP(nidof, temporary)
npred†	

Output:

IU(nidof+1)	JU(ncoef2)
ncoef2	iflag‡

† This is the predicted value of ncoef2.

‡ Return 0 if operation performs successfully. Otherwise, return the row id where the code stops.

7. Numerical factorization of Kii (NUMFA1 subroutine for a symmetrical problem or UNSYMNUMFA1 subroutine for an unsymmetrical problem)

Description: Numerical factorization of K_{ii}

Argument list: nidof, IA, JA, AD, AN, IU, JU, DI, UN, IP, IUP, ISUPD, AN2, UN2, DI2

Input:

nidof
IA(nidof+1)
JA(IA(nidof+1)-1)
AD(nidof)
AN(IA(nidof+1)-1)
IP(nidof)
IUP(nidof)
ISUPD(nidof)
AN2(IA(nidof+1)-1)

Output:

IU(nidof+1)
JU(ncoef2)
DI(nidof)
UN(ncoef2)
UN2(ncoef2)
DI2(nidof)

8. Solving for boundary DOFs displacements (CDDBDOF subroutine)

Description: Using an iterative solver to solve for boundary degrees of freedom

Argument list: me, np, iter, isy, islvr, XB, ibgen, maxiter, ngbjdof, nbdof, nidof, BI, BB, IU, JU, DI, UN, UN2, ISUPD, iopf, IABB, JABB, ANBB, ANBB2, ADBB, IABBC, JABBC, ANBBC, IABI, JABI, ANBI, ANBI2, IAIB, JAIB, ANIB, IAII, JAII, ANII, ANII2, ADII, IBDOFARE, IDDPAR, errtol, TIME, memused, memmax, memavai, iexceed, ifin1, ma, io3

Input:

me
np
iter
isy
islvr
ibgen
maxiter
ngbjdof
nbdof
nidof
BI(nidof)
BB(nbdof)
IU(nidof+1)
JU(ncoef2)
DI(nidof)
UN(ncoef2)
UN2(ncoef2)
ISUPD(nidof)
iopf
IABB(nbdof+1)
JABB(IABB(nbdof+1)-1)
ANBB(IABB(nbdof+1)-1)
ANBB2(IABB(nbdof+1)-1)
ADBB(nbdof)
IABBC(nbdof+1)
JABBC(IABBC(nbdof+1)-1)
ANBBC(IABBC(nbdof+1)-1)
IABI(nbdof+1)
JABI(IABI(nbdof+1)-1)
ANBI(IABI(nbdof+1)-1)
ANBI2(IABI(nbdof+1)-1)
IAIB(nidof+1)

JAIB(IAIB(nidof+1)-1)
IAII(nidof+1)
ANII(IAII(nidof+1)-1)
ADII(nidof)
IDDPAR(200)
TIME(30)
memmax
iexceed*
ANIB(IAIB(nidof+1)-1)
JAII(IAII(nidof+1)-1)
ANII2(IAII(nidof+1)-1)
IBDOFARE(nbdof)
errtol
memused
memavai
ifin1**

Output:

XB(ngbjdof)

* Return 0 if memory does not exceed in iterative solver subroutine.

** Return 0 if iterative solver fails to find the boundary displacements

9. Solving for interior DOFs displacement (ZIR subroutine)

Description: Using boundary DOF displacements obtained from previous to find interior DOF displacements

Argument list: XB, IABI, JABI, ANBI, FI, IU, JU, DI, UN, XI, nidof, ncoef2, ngbjdof, nbdof, IBDOFARE, TEMP01, IAIB, JAIB, ANIB, iter, isy, ANBI2, UN2

Input:

XB(ngbjdof)
JABI(IABI(nbdof+1)-1)
FI(nidof)
JU(IU(nidof+1)-1)
UN(IU(nidof+1)-1)
ncoef2
nbdof
TEMP01(nbdof, temporary)
JAIB(IAIB(nidof+1)-1)
iter
ANBI2(IAIB(nidof+1)-1)
IABI(nbdof+1)
ANBI(IABI(nbdof+1)-1)
IU(nidof+1)
DI(nidof)
nidof
ngbjdof
IBDOFARE(nbdof)
IAIB(nidof+1)
ANIB(IAIB(nidof+1)-1)
isy
UN2(IU(nidof+1)-1)

Output:

XI(nidof)

10. Revert displacements to the original (COMBDISP subroutine)

Description: Revert displacements on all processors to the original system

Argument list: me, np, nbdof, nidof, INVP, XB, XI, XG, NGBMAP, ndofpn,

IPERM, IBDOFARE, ndofall, nodesall

Input:

me
nbdof
INVP(nidof)
XI(nidof)
ndofpn
IBDOFARE(nbdof)
nodesall
np
nidof
XB(nbdof)
NGBMAP(nodes)
IPERM(nidof)
ndofall

Output:

XG(ndofall)

B.2 LIST OF VARIABLES

Name	Size	Type	Description
ADBB	nbdof	complex*16	An array containing diagonal information of $K_{bb}^{(r)}$ matrix (see chapter 3.1)
ADII	nidof	complex*16	An array containing diagonal information of $K_{ii}^{(r)}$ matrix (see chapter 3.1)
ANBB	IABB(nbdof+1)-1	complex*16	An array containing the values of non-zero terms of the upper triangular part of $K_{bb}^{(r)}$ matrix (see chapter 3.1)
ANBB2	IABB(nbdof+1)-1	complex*16	An array containing the values of non-zero terms of the lower triangular part of $K_{bb}^{(r)}$ matrix (see chapter 3.1)
ANBBC	IABBC(nbdof+1)-1	complex*16	An array containing the values of non-zero terms of $K_{bb}^{(r)}$ matrix (upper, lower and diagonal parts of the matrix)
ANBI	IABI(nbdof+1)-1	complex*16	An array containing the values of non-zero terms of $K_{bi}^{(r)}$ matrix (see chapter 3.1)
ANBI2	IABI(nbdof+1)-I	complex*16	An array containing the values of non-zero terms of $K_{ib}^{(r)}$ matrix (compressed storage scheme in column, CSC, format)
ANIB	IABI(nbdof+1)-1	complex*16	An array containing the values of non-zero terms of $K_{ib}^{(r)}$ matrix (CSR format) (see chapter 3.1)
ANII	IAII(nidof+1)-1	complex*16	An array containing the values of non-zero terms of the upper triangular part of $K_{ii}^{(r)}$ matrix (see

Name	Size	Type	Description
			chapter 3.1)
ANII2	IAII(nidof+1)-1	complex*16	An array containing the values of non-zero terms of the lower triangular part of $K_{ii}^{(r)}$ matrix (see chapter 3.1)
ANIIC	IAIIC(nidof+1)-1	complex*16	An array containing the values of non-zero terms of $K_{ii}^{(r)}$ matrix (upper, lower and diagonal parts of the matrix)
BB	nbdof	complex*16	A complex array containing the magnitude of external loads applied on each boundary dof in the subdomain
BI	nidof	complex*16	A complex array containing the size of external loads applied on each interior dof in the subdomain
CMPC	IAMPC(nmpc+1)-1	complex*16	An array containing coefficient value of dofs in JAMPC array for each MPC equation of the subdomain (see step 13, Chapter 6.3)
CMPCG	IAMPCG(nmpcg+1)-1	complex*16	An array containing coefficient value of dofs in JAMPCG array for each MPC equation of the domain (see Appendix A.1)
CT1	varied	complex*16	A complex temporary array with varied size.
CT2	varied	complex*16	A complex temporary array with varied size.
CT3	varied	complex*16	A complex temporary array with varied size.
CT4	varied	complex*16	A complex temporary array with varied size.
CT5	varied	complex*16	A complex temporary array with varied size.
CT6	varied	complex*16	A complex temporary array with varied size.
CT7	varied	complex*16	A complex temporary array with varied size.

Name	Size	Type	Description
DB	nbdof	complex*16	An array containing the diagonal information of the factorized $K_{bb}^{(r)}$ matrix
DI	nidof	complex*16	An array containing the diagonal information of the factorized $K_{ii}^{(r)}$ matrix
DB2	nbdof	complex*16	A working array used in unsymmetrical factorizing subroutine
DI2	nidof	complex*16	A working array used in unsymmetrical factorizing subroutine
errtol		real*8	error tolerance used in the iterative solver
FFG	nloadofg	complex*16	The size of the external loads applied on the domain
IABB	nbdof+1	integer	An integer array containing the information of non-zero terms in the upper triangular part of $K_{bb}^{(r)}$ in CSR format (see chapter 3.1)
IABBC	nbdof+1	integer	An integer array containing the information of non-zero terms of $K_{bb}^{(r)}$ in CSR format (upper, lower and diagonal parts of the matrix)
IABI	nbdof+1	integer	An integer array containing the information of non-zero terms of $K_{bi}^{(r)}$ in CSR format (see chapter 3.1)
IAIB	nidof+1	integer	An integer array containing the information of non-zero terms of $K_{ib}^{(r)}$ in CSR format (see chapter 3.1)
IAII	nidof+1	integer	An integer array containing the information of

Name	Size	Type	Description
			non-zero terms in the upper triangular part of $K_{bb}^{(r)}$ in CSR format (see chapter 3.1)
IAIIC	nidof+1	integer	An integer array containing the information of non-zero terms of $K_{ii}^{(r)}$ in CSR format (upper, lower and diagonal parts of the matrix)
IAMPC	nmpc+1	integer	An array containing the starting location in JAMPC array of each MPC equation of the subdomain (see step 13, Chapter 6.3)
IAMPCG	nmpcg+1	integer	An array containing the starting location in JAMPCG array of each MPC equation of the domain (see Appendix A.1)
IBCB	nbdof	integer	an integer array storing the list of boundary dofs associated with Dirichlet boundary conditions (0 if not associated with Dirichlet boundary dof and 1 if associated with Dirichlet boundary dof)
IBCI	nidof	integer	an integer array storing the list of interior dofs associated with Dirichlet boundary conditions (0 if not associated with Dirichlet boundary dof and 1 if associated with Dirichlet boundary dof)
IBDOFARE	nbdof	integer	A mapping array between local boundary nodes and global boundary nodes where IBDOFARE(local boundary node ID) returns global boundary node ID
ibgen		integer*8	Preconditioning scheme used in iterative solver (see details in Chapter 2.2)
IDDPAR	200	integer*8	An integer array containing problem and

Name	Size	Type	Description
			program information (See Appendix A for more information)
IE	nsizeiea	integer	Distributed element connectivity information (see Chapter 6.1 for more details)
IEBB	nels+1+nmpc	integer	Element connectivity information (real elements + artificial elements from MPC equations) associated with the boundary nodes
IEII	nels+1+nmpc	integer	Element connectivity information (real elements + artificial elements from MPC equations) associated with the interior nodes
ielem		integer*8	Acoustic parameter
IELMAP	nel	integer	Used with NCHK array in breaking phase to index interior elements and boundary elements in all subdomains. Boundary elements are stored in IELMAP array from NCHK(nsub+1) to (NCHK(nsub+2)-1), while interior elements of i^{th} subdomain are stored from NCHK(i) to NCHK(i+1)-1 (see Chapter 6.3 for details)
IELMAPMPC	nmpcg	integer	Used with NCHKMPC array in breaking phase to index interior artificial elements and boundary artificial elements in all subdomains. Boundary artificial elements are stored in IELMAPMPC array from NCHKMPC(nsub+1) to (NCHKMPC(nsub+2)-1), while interior artificial elements of i^{th} subdomain are stored from NCHKMPC(i) to NCHKMPC(i+1)-1.

Name	Size	Type	Description
IELOWNER	nel	integer*2	Used in breaking phase where abs(IELOWNER(i)) indicates subdomain the i^{th} element belongs to. The minus sign of IELOWNER(i) indicates the boundary element.
ierr		integer	MPI error return flag
iexceed		integer*8	Flag used in iterative solver to check if the memory required in the solver is more than the memory the processor has
ifin		integer*8	Set to be 0 before calling the cddfea subroutine. Upon the exit of the subroutine, ifin is 1 if the code successfully get the result and 0 otherwise.
iflag		integer*8	System flag indicating the error in MPC breaking (input = 0). Upon success of this part, there is an error in MPC breaking if iflag is not zero
ifreq		integer*8	Acoustic Parameter
IIX	nels	integer	An array indicating the location of elements along x direction. It is used in 3D acoustic problem.
IIY	nels	integer	An array indicating the location of elements along y direction. It is used in 3D acoustic problem.
IIZ	nels	integer	An array indicating the location of elements along z direction. It is used in 3D acoustic problem.
imajor		integer*8	Scheme used to select the owner of the elements. 0 = The owner of the first node is the owner of

Name	Size	Type	Description
			the element. 1 = The owner of the element is the processor owns the majority of the nodes in that element.
imajormpc		integer*8	Scheme used to select the owner of the artificial elements. 0 = The owner of the first node is the owner of the artificial element. 1 = The owner of the artificial element is the processor owns the majority of the nodes in that element.
IMPCOWNER	nmpcg	integer*2	Used in breaking phase where IMPCOWNER(i) indicates subdomain the i^{th} artificial element belongs to. The minus sign of IMPCOWNER(i) indicates the boundary element.
INVP	nbdor or nidof	integer	Upon the successful of the METIS reordering part, this is an array storing the inverse-permutation of the permuted matrix. The size of the array will be nbdof if only 1 processor is used to solve the problem, and nidof, otherwise (see Chapter 3.2 for more information)
io3		integer*8	Output unit for timing and problem information of the process
io7		integer*8	Output unit for timing and problem information of all the processes
iopf		integer*8	An integer number indicated the operation counts in Nguyen's direct solver.
ipartieje		integer*8	Element connectivity partitioning scheme (see Chapter 6.2 for details)

Name	Size	Type	Description
IPERM	nbdof or nidof	integer	Upon the successful of the METIS reordering part, this is an array storing the permutation of the permuted matrix. The size of the array will be nbdof if only 1 processor is used to solve the problem, and nidof, otherwise (see Chapter 3.2 for more information)
iprob		integer*8	Problem type
ireord		integer*8	An integer used to specify the reordering scheme used in the code (0: no reordering, 1: METIS reordering)
IRITE	20	integer	An integer array used to specify the level of information returned during the execution
islvr		integer*8	An integer number specified the type of direct solver used to factorized the coefficient matrix (1: Duc Nguyen's solver)
ISUPD	nbdof or nidof	integer	An array storing supernode information
isy		integer*8	An integer number specified the type of the coefficient whether or not it is symmetric. (0: unsymmetrical matrix, 1: symmetrical matrix
IT1	varied	integer	An integer, temporary array with varied size
IT10	varied	integer	An integer, temporary array with varied size
IT11	varied	integer	An integer, temporary array with varied size
IT2	varied	integer	An integer, temporary array with varied size
IT3	varied	integer	An integer, temporary array with varied size
IT4	varied	integer	An integer, temporary array with varied size
IT5	varied	integer	An integer, temporary array with varied size

Name	Size	Type	Description
IT6	varied	integer	An integer, temporary array with varied size
IT7	varied	integer	An integer, temporary array with varied size
IT8	varied	integer	An integer, temporary array with varied size
IT81	varied	integer*8	An integer, temporary array with varied size
IT9	varied	integer	An integer, temporary array with varied size
iter		integer*8	Type of the solver used to solve for displacements of boundary dofs 0: direct solver (not yet implemented) Symmetrical problem: 1-3: Preconditioned Conjugate Gradient Unsymmetrical problem: 2: GMRES 3: FGMRES
IU	nbdof+1 (1 CPU) nidof+1 (>1 CPU)	integer*8	An integer array containing the information of non-zero terms in the upper triangular part of the factorized $K_{bb}^{(r)}$ or the factorized $K_{ii}^{(r)}$ in CSR format (used with JU)
iway		Integer*8	Domain breaking scheme (0: ParMETIS, 1: Author's scheme),
JABB	IABB(nbdof+1)-1	integer	An integer array containing the information of non-zero terms in the upper triangular part of $K_{bb}^{(r)}$ in CSR format
JABBC	IABBC(nbdof+1)-1	integer	An integer array containing the information of non-zero terms of $K_{bb}^{(r)}$ in CSR format (upper, lower and diagonal parts of the matrix)

Name	Size	Type	Description
JABI	IABI(nbdof+1)-1	integer	An integer array containing the information of non-zero terms of $K_{bi}^{(r)}$ in CSR format
JAIB	IAIB(nidof+1)-1	integer	An integer array containing the information of non-zero terms of $K_{ib}^{(r)}$ in CSR format
JAII	IAII(nidof+1)-1	integer	An integer array containing the information of non-zero terms in the upper triangular part of $K_{ii}^{(r)}$ in CSR format
JAIIC	IAIIC(nidof+1)-1	integer	An integer array containing the information of non-zero terms of $K_{ii}^{(r)}$ in CSR format (upper, lower and diagonal parts of the matrix)
JAMPC	IAMPC(nmpc+1)-1	integer	List of dofs associated with each MPC equations (see step 13, Chapter 6.3)
JAMPCG	IAMPCG(nmpcg+1)-1	integer	List of dofs associated with each MPC equations (see Appendix A.1)
JE	nsizejea	integer	Distributed element connectivity information
JEBB	IEBB(nels+1)-1	integer	Subdomain's element connectivity information associated with the boundary nodes
JEII	IEII(nels+1)-1	integer	Subdomain's element connectivity information associated with the interior nodes
JU	1 CPU: IU(nbdof+1)-1 Multi CPUs: IU(nidof+1)-1	integer	An integer array containing the information of non-zero terms in the upper triangular part of the factorized $K_{bb}^{(r)}$ or the factorized $K_{ii}^{(r)}$ in CSR format
LOADOFSG	nloadofg	integer	List of dofs associated with the external loads applied on the entire domain

Name	Size	Type	Description
ma		integer*8	2D acoustic parameter
maxiter		integer*8	Maximum iterations limit for the iterative solver
me		integer	Processor ID
MEMATER	nel	integer	The material properties set id of each element
memavai		integer*8	Amount of memory available, in bytes, for each processor.
memmax		integer*8	Amount of maximum memory used, in bytes, during the execution of the code.
memused		integer*8	This variable will keep track of amount of memory used, in bytes, during the execution
MET	node	integer	Result from ParMETIS; MET(i) indicates the owner of i^{th} node of the domain
na		integer*8	2D acoustic parameter
nbbcomb		integer*8	Number of non-zero terms in $K_{bb}^{(r)}$ matrix (upper, lower and diagonal parts of the matrix).
nbc		integer*8	Number of Dirichlet boundary conditions
NBCDOF	nbc	integer	List of dofs associated with subdomain's Dirichlet boundary conditions
NBCDOFS	ndir	integer	List of dofs associated with entire domain's Dirichlet boundary conditions
nbdof		integer*8	Number of subdomain's boundary dofs
nbj		integer*8	Number of subdomain's boundary nodes
nbjall		integer*8	Number of total boundary nodes of the entire domain
NCHK	nsub+2	integer	Used with IELMAP array in breaking phase to index interior elements and boundary elements in

Name	Size	Type	Description
			all subdomains. Boundary elements are stored in IELMAP array from NCHK(nsub+1) to (NCHK(nsub+2)-1), while interior elements of i^{th} subdomain are stored from NCHK(i) to NCHK(i+1)-1.
NCHKMPC	nsub+2	integer	Used with IELMAPMPC array in breaking phase to index interior artificial elements and boundary artificial elements in all subdomains. Boundary artificial elements are stored in IELMAPMPC array from NCHKMPC(nsub+1) to (NCHKMPC(nsub+2)-1), while interior artificial elements of subdomain i are stored from NCHKMPC(i) to NCHKMPC(i+1)-1.
ncoef		integer*8	Acoustic parameter
ncoef1bb		integer*8	Number of non-zero terms in the upper triangular part of $K_{bb}^{(r)}$ matrix, which is the same as non-zero terms in the lower triangular part of the matrix
ncoef1bi		integer*8	Number of non-zero terms in $K_{bi}^{(r)}$ and $K_{ib}^{(r)}$ matrices.
ncoef1ii		integer*8	Number of non-zero terms in the upper triangular part of $K_{ii}^{(r)}$ matrix, which is the same as non-zero terms in the lower triangular part of the matrix
ncoef2bb		integer*8	Number of non-zero terms in the upper triangular

Name	Size	Type	Description
			part of the factorized $K_{bb}^{(r)}$ matrix
ncoef2ii		integer*8	Number of non-zero terms in the upper triangular part of the factorized $K_{ii}^{(r)}$ matrix
ncoef2temp		integer*8	Estimated number of non-zero terms of the factorized coefficient matrix. This number is used to allocate the number of temporary arrays used in factorization phase.
NCUM	neltype+1	integer	An array containing the number of elements of each element type for the entire domain. NCUM(i+1)-NCUM(i) indicates the number of elements of i^{th} element type
NCUMS	neltype+1	integer	An array containing the number of elements of each element type in the subdomain. NCUMS(i+1)-NCUMS(i) indicates the number of elements of i^{th} element type
ndir		integer*8	Number of subdomain Dirichlet boundary conditions
ndofall		integer*8	Number of dofs in the entire domain
ndofalls		integer*8	Number of dofs in the subdomain
ndofpe		integer*8	Number of dofs per element
ndofpn		integer*8	Number of dofs per node
nel		integer*8	Number of elements in the entire domain
nels		integer*8	Number of elements in the subdomain
neltype		integer*8	Number of element types
nep		integer*8	Expected sized of adjacency array used in reordering phase

Name	Size	Type	Description
nexpect		integer*8	Expected ratio of non-zero terms after factorization and before factorization (ncoef2/ncoef1)
ngbjdof		integer*8	Number of total boundary dofs in the entire domain
NGBMAP	nodes	integer	Mapping of the subdomain nodes and the global original node of the domain (ngbmap(local node ID) = global node ID)
nidof		integer*8	Number of interior dofs in the subdomain
niicomb		integer*8	Number of non-zero terms in $K_{ii}^{(r)}$ matrix (upper+lower+diagonal).
NKBITYPE	neltype+2	integer	Temporary array used in the phase to obtain element connectivity of the subdomain
nloadof		integer*8	Number of the external loads applied on the subdomain
nloadofg		integer*8	Number of external loads applied on the entire domain
nmpc		integer*8	Number of subdomain MPC equations
nmpcg		integer*8	Number of Multi-point constraint equations of the entire domain
nnx		integer*8	3D acoustic parameter
nny		integer*8	3D acoustic parameter
nnz		integer*8	3D acoustic parameter
node		integer*8	Number of nodes in the entire domain
NODE1	nels	integer	List of the first node of the subdomain elements
NODE2	nels	integer	List of the second node of the subdomain

Name	Size	Type	Description
			elements
NODE3	nels	integer	List of the third node of the subdomain elements
NODE4	nels	integer	List of the fourth node of the subdomain elements
NODE5	nels	integer	List of the fifth node of the subdomain elements
NODE6	nels	integer	List of the sixth node of the subdomain elements
NODE7	nels	integer	List of the seventh node of the subdomain elements
NODE8	nels	integer	List of the eighth node of the subdomain elements
nodes		integer*8	Number of nodes in the subdomain
noffiejea		integer*8	offset of JE after partitioning (see Chapter 6.1)
np		integer	Number of processors
npe		integer*8	Nodes per element
npropmat		integer*8	Number of material properties sets
nsizeiea		integer*8	Size of IE (see chapter 6.1)
nsizejea		integer*8	Size of JE (see chapter 6.1)
nsub		integer	Number of subdomains (i.e. number of processors)
nsub2		integer	Number of subdomains (i.e. number of processors)
nts		integer*8	Number of elements and MPC equations associated with the subdomain
numater		integer	reserved for future use
PROP	max(nnx*nny+1,200)	complex*16	An array containing the material properties values of all material sets

Name	Size	Type	Description
PROPR	100	real*8	An array containing the material properties values of all material sets
RMPC	nmpc	complex*16	MPC equations information array of the subdomain (see Chapter 6.3)
RMPCG	nmpcg	complex*16	RHS array of MPC equations (see Appendix A.1)
STATUS	MPI_STATUS_SIZE	integer	MPI variable
TIME	30	real*8	A double precision array containing times of each step
UN	1 CPU: IU(nbdof+1)-I Multi CPUs: IU(nidof+1)-1	complex*16	An array containing the values of non-zero terms of the upper triangular part of the factorized matrix.
UN2	1 CPU: IU(nbdof+1)-1 Multi CPUs: IU(nidof+1)-1	complex*16	An array containing the values of non-zero terms of the upper triangular part of the factorized matrix.
wn		real*8	Acoustic parameter
wy		real*8	Acoustic parameter
XB	nbdof	complex*16	An array containing the subdomain boundary displacements.
XCOORG	node	real*8	A double precision array storing the x coordinates values of each node. Please note that, for a 2D acoustic problem, the size of this array is ma.
XI	nidof	complex*16	An array containing the subdomain interior displacements.

Name	Size	Type	Description
xmach		real*8	Acoustic parameter
XSOL	node*ndofpn	complex*16	An array containing the entire domain displacements (i.e. solution vector)
YCOORG	node	complex*16	A double precision array storing the y coordinates values of each node. Please note that, for a 2D acoustic problem, this array is not used.
ZCOORG	node	complex*16	A double precision array storing the z coordinates values of each node. Please note that, for a 2D acoustic problem, the size of this array is na.
LOADOFS			
FF			
IELIST	Number of elements attached to the processor's nodes during ParMETIS phase	integer	The list of elements attached to the processor's nodes during ParMETIS phase
MYIE	Number of elements attached to the processor's nodes during ParMETIS phase+1	integer	Element connectivities of the element attached to the processor's nodes during ParMETIS phase
MYJE	The value of last element of MYIE array – 1	integer	Element connectivities of the element attached to the processor's nodes during ParMETIS phase

APPENDIX C

SOURCE CODES AND INPUT/OUTPUT FILES

Source codes and input/output files of this work are available upon request. Please send an email to Siroj Tungkahotara (toohtaah@gmail.com) or Prof. Duc T. Nguyen (dnguyen@odu.edu).

APPENDIX D

DATA FOR 3-D SYMMETRICAL ACOUSTIC EXAMPLE WITH 40 MPC EQUATIONS

Based on the input format explained in Appendix A.1, the information about 40 MPC equations used in example 5.3 are presented below.

iampcg(-) = 1, 4, 6, 10, 12, 15, 17, 19, 22, 25, 29, 32, 34, 36, 38, 40, 43, 48, 50, 52, 54, 56, 59, 61, 63, 66, 68, 70, 72, 74, 76, 79, 81, 83, 85, 87, 90, 92, 94, 96, 101

rmpcg(-) = (1.113,-2.542), (-0.058, 0.543), (-3.482, 0.552), (1.329,-1.575), (1.156,-3.515), (2.118, 3.811), (3.097, 2.203), (-5.661, 1.112), (-0.197, 3.658), (-3.675, 4.089), (5.201,-1.248), (-1.489,-6.217), (-3.535,-5.196), (-1.379,-0.263), (4.672, 4.663), (1.943, 1.327), (1.085,-9.044), (0.713,-4.634), (2.174, 1.490), (-2.978, 5.488), (0.480,-0.395), (-3.254, 2.710), (2.384, 0.105), (-4.440,-0.730), (0.565,-0.772), (-1.401, 0.392), (-6.105,-0.011), (-1.742, 0.976), (1.532,-5.268), (-1.533,-2.417), (-3.555,-1.979), (-3.741, 0.413), (-2.560,-3.965), (0.909, 4.926), (-3.986,-1.119), (-0.826, 6.011), (0.734, 2.230), (-4.553,-2.019), (7.272,-1.360), (-0.782,-0.236)

jampcg(-) = 787157, 792828, 1619569, 1873504, 853376, 1633969, 289645, 409288, 769038, 1228959, 1821994, 1714071, 1685447, 1145252, 557195, 1252008, 1067513, 1158428, 1324896, 1740968, 82835, 889494, 1267175, 1789492, 185176, 744539, 623590, 812257, 231147, 413392, 1487666, 801145, 497459, 1557438, 1655569, 1376091, 103296, 1759068, 797658, 1936850, 371303, 71898, 1028061, 1566422, 95893, 1599124, 1794188, 682287, 1176487, 1580508, 1416150, 1513018, 37429, 731461, 1530237, 855888, 647956, 1063340, 1142547, 790094, 167233, 229032, 1708155, 448360, 1233927, 196830, 417431, 406875, 1927723, 1359731, 442714, 1668188, 832988, 1521609, 1979731, 1720064, 1504462, 397546, 1919270, 783609, 422891, 187568, 1866222, 509187, 766798, 406021, 428021, 826304, 1645017, 236282, 1114086, 1763557, 176016, 904780, 9601, 1650610, 826211, 1138416, 967633,

156591

cmpcg(1) = (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0), (1d0,-2d0), (-3d0,4d0), (-1d0,1d0), (3d0,-2d0), (4d0,1d0), (0.7d0,0.6d0), (0.2d0,-0.5d0), (3d0,-2d0), (1d0,-3d0), (0.36d0,0.73d0)

VITA

Siroj Tungkahotara was born in Bangkok, Thailand on April 26, 1975. After his graduation with a Bachelor of Civil Engineering from the faculty of engineering, Chulalongkorn University, Thailand, he worked as a structural design engineer for Italian-Thai Development PCL for two years. Then in 2000, he attended the department of Civil and Environmental Engineering, Old Dominion University and earned a Master of Engineering in 2001. Later in the same year, he joined the Ph.D. program at the university and became a Ph.D. candidate in 2004. During his academic study, he worked on several projects granted by the National Aeronautics and Space Administration (NASA) at Langley Research Center and published many articles as follow:

Nguyen, D. T. and S. Tungkahotara. "Finite Element Parallel-Sparse Computation for Engineering/Science Applications." An invited keynote lecture at the International Conference on Smart Materials and Adaptive Structures: Mathematical Modeling and Computation. Tangier, Morocco, 14-16 April 2008.

Nguyen, D. T. and Siroj Tungkahotara. "Parallel-Vector/Cache Algorithms/Software for Large-Scale Computation." Proceedings of the 2002 Space Radiation Shielding Technology Workshop. Hampton, VA: NASA Langley Research Center, April 4, 2002.

Nguyen, D. T., et al. "Parallel Finite Element Domain Decomposition for Structural/Acoustic Analysis." 2003: 189-201.

Nguyen, D. T., S. Tungkahotara and L. Azrar. "A Stand-Alone Nonlinear BFGS Sparse Equation Solver Algorithms and Software with Parallel Domain Decomposition Capability." Proceedings of the International Conference on Smart Materials and Adaptive Structures: Mathematical Modeling and Computation. Tangier, Morocco, April 14-16, 2008.

Nguyen, D.T., et al. "Domain Decomposition Formulation and Implementation for Finite Element Structural/Acoustic Analysis under Parallel MPI/Fortran Computer Environments." Proceedings of the 9th International Conference on Numerical Methods and Computational Mechanics. Miskolc, Hungary: University of Miskolc, July 15-19, 2002.

Nguyen, E. N.D., et al. "Comparing DNA Sequences by Dynamic Programming in Sequential and Parallel Computer Environments." Presented at the "2006 WSEAS International Conference on Mathematical Biology and Ecology" (MABES' 06). Miami, FL, Jan. 18-20, 2006.

—. "MPI Parallel FORTRAN Dynamic Programming Algorithms and Software for Comparisons of DNA Sequences." WSEAS Transactions on Biology and Biomedicine February 2006: 145-152.

Tungkahotara, S., et al. "Simple and Efficient Parallel Dense Equation Solvers." Proceedings of the 9th International Conference on Numerical Methods and Computational Mechanics. Miskolc, Hungary: University of Miskolc, July 15-19, 2002.